A Survey of Smart Contract Formal Specification and Verification

PALINA TOLMACH, Nanyang Technological University, Singapore and Institute of High Performance Computing (A*STAR), Singapore
YI LI, SHANG-WEI LIN, and YANG LIU, Nanyang Technological University, Singapore
ZENGXIANG LI, Institute of High Performance Computing (A*STAR), Singapore

A smart contract is a computer program that allows users to automate their actions on the blockchain platform. Given the significance of smart contracts in supporting important activities across industry sectors including supply chain, finance, legal, and medical services, there is a strong demand for verification and validation techniques. Yet, the vast majority of smart contracts lack any kind of formal specification, which is essential for establishing their correctness. In this survey, we investigate formal models and specifications of smart contracts presented in the literature and present a systematic overview to understand the common trends. We also discuss the current approaches used in verifying such property specifications and identify gaps with the hope to recognize promising directions for future work.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Formal software verification**; **Specification languages;**

Additional Key Words and Phrases: Smart contract, formal verification, formal specification, properties

ACM Reference format:
Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (June 2021), 38 pages.
https://doi.org/10.1145/3464421

1 INTRODUCTION

A proposal to store timestamped tamper-resistant information in cryptographically secure chained blocks has gradually evolved from 1991 [82] into what we now know as *blockchain*: a distributed ledger shared between nodes of a peer-to-peer network following a certain consensus protocol, and

This research is partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (Award No. 2018-T1-002-069) and Tier 2 (Award No. MOE2018-T2-1-068), and by the National Research Foundation, Singapore, and the Energy Market Authority, under its Energy Programme (EP Award No. NRF2017EWT-EP003-023). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore and the Energy Market Authority.
Authors' addresses: P. Tolmach, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798 and Institute of High Performance Computing (A*STAR), 1 Fusionopolis Way, #16-16 Connexis North, Singapore, 138632; email: palina001@ntu.edu.sg; Y. Li, S.-W. Lin, and Y. Liu, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; emails: {yi_li, shang-wei.lin, yangliu}@ntu.edu.sg; Z. Li, Institute of High Performance Computing (A*STAR), 1 Fusionopolis Way, #16-16 Connexis North, Singapore, 138632; email: zengxiang_li@outlook.com.

0360-0300/2021/06-ART148 $15.00
https://doi.org/10.1145/3464421

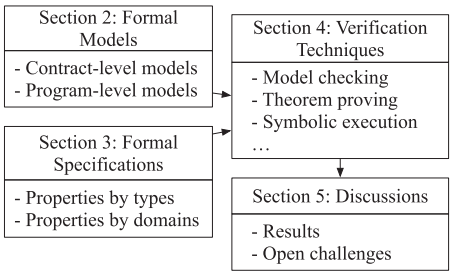

Fig. 1. Organization of the article.

```
("formal"
  ("verification" OR "modeling" OR "specification")
  "of smart contracts")
OR ("smart contract properties")
OR ("smart contract temporal properties")

("formal"
  ("verification" OR "modeling" OR "specification")
  "of smart contracts")
AND ("properties" OR ("temporal properties"))
```

Fig. 2. Search queries used in Google Scholar.

one of the fastest-growing technologies of the modern time. The blockchain technology owes its initial popularity to Bitcoin [126], which appeared in 2008. Later, Ethereum [179] started the era of Blockchain 2.0 and expanded the capabilities and applications of blockchain by introducing Turing-complete *smart contracts*. The term *smart contract* was originally proposed by Nick Szabo [163], who suggested encoding the contractual terms and protocols of the involved parties in the digital form. In the blockchain context, smart contracts currently refer to computer programs executed on top of blockchain, and we concentrate on this definition of a smart contract in this work. In Ethereum, each contract is associated with an address, a balance, and a piece of executable code. The functions in a contract can be invoked via transaction messages sent from platform users. Most of the smart contracts are written in Turing-complete languages, such as Solidity [6].

The market cap of the Ethereum cryptocurrency was worth 20 billion as of February 1, 2020, and several hundred thousand transactions are processed each day [5]. Unfortunately, the adoption of blockchain and smart contracts is also accompanied by severe attacks, often due to domain-specific security pitfalls in the smart contract implementations. Examples include theft of Ether through notorious attacks on the DAO [155] and Parity Multisig Wallet [103], as well as the freezing of users' funds in the same Parity Multisig Wallet [164] and the King of the Ether Throne [1] smart contracts. Extensive research has been done over the past several years—by technology giants, non-profit organizations, and the research community—on the techniques for verifying smart contract correctness. In this survey, we review the recent advances in the applications of formal methods in the analysis of smart contracts, with a focus on the domain-specific properties and various formalisms supporting the specification and verification of such properties.

We organize the article following a structure outlined in Figure 1. *Formal verification* proves or disproves the correctness of a system by checking the *formal (mathematical) model* of the system against a certain *formal specification*. A specification is a set of *properties* describing the desired behaviors of a smart contract, usually defined by developers' intention. Models and specifications can be defined at different levels of abstraction via various types of formalisms, which we overview in Sections 2 and 3, respectively. Section 3 additionally describes a taxonomy of smart contract properties, which highlights the emerging trends in the literature. Section 4 closes the loop by reviewing formal verification techniques utilized to prove the correctness of smart contract models with respect to the desired properties. Section 5 concludes the survey with our observations on major trends, challenges, and future directions in the formal specification and verification of smart contracts.

1.1 Methodology

Sections 1.1 and 1.2 outline the methodology that we followed to extensively collect the existing work related to smart contract formal specification and verification, and an overview of the body of related works, respectively.

Research Scope. This article is aimed to overview, analyze, and classify approaches to formal modeling of smart contracts, specification of smart contract properties, and techniques employed in the verification of those. The scope of the survey includes a study of common property specifications of smart contracts, characterizing their functional correctness and security guarantees. The focus of the survey is narrowed down to smart contract functional behavior and does not embody problems related to other blockchain-related execution aspects, such as scalability, consensus, interaction with IoT systems, and so on. Therefore, the purpose of the article is to provide answers to the following *research questions*:

RQ1: What are the formal techniques used for modeling, specification, and verification of smart contracts? What are the common formal requirements specified and verified by these techniques?

RQ2: What are the challenges introduced by smart contract and blockchain environment in formalizing and verifying smart contracts?

RQ3: What are the current limitations in smart contract formal specification and verification and what research directions may be taken to overcome them?

Search Criteria. To provide a complete survey covering most of the publications related to smart contract specification and verification since 2014, we created a publication repository that includes 202 papers published from September 2014 to June 2020. To collect the relevant literature for analysis, we have performed the search in Google Scholar, ACM, IEEE, and Springer databases using the two queries shown in Figure 2.

In October 2019, we have explored 10 pages of the results retrieved for each query and obtained 70 papers that fall into the scope of the survey and have been published since 2014. Based on the collected set of papers, we performed snowballing and inverse snowballing to obtain a more comprehensive view by collecting 85 more publications. Additionally, to provide an overview of the state-of-the-art, we included another 47 recently published papers that were retrieved using the two aforementioned queries from October 2019 to June 2020 by Google Scholar at the time of writing. Overall, we collected and studied 202 relevant papers. The majority of them were published in the proceedings of 70 conferences and workshops (including ACM CCS, ASE, ESEC/FSE, IEEE SANER, S&P, IEEE/ACM ICSE, NDSS, ISoLA, FC, FM, and the affiliated workshops) or one of 13 journals (including *ACM TOSEM, IEEE TSE,* and *IEEE Access*). Using our search methodology, we additionally identified 53 articles that conduct a survey on smart contract types, languages, vulnerabilities, and verification techniques, or propose security and design patterns for their development. We describe the collected articles in an online repository: https://ntu-srslab.github.io/smart-contract-publications/.

In the presented work, we do not claim to provide a complete set of properties that smart contracts are supposed to meet. However, by means of the presented analysis and classification, we seek to provide a firm formalized foundation and a stable structure for smart contract property specifications, what we hope will facilitate the development of correct, secure, and standardized smart contracts.

1.2 Related Works

By now, a multitude of surveys addressing smart contract analysis has been published. These include a review of security vulnerabilities [23, 79, 148, 182], verification approaches and tools [48, 59, 62, 136, 139], formal specification and modeling techniques [32, 94, 156], and languages for smart contract development [85, 134]. We found several pieces of work discussing common design patterns of smart contracts [178, 181], as well as studies of smart contract platforms and practical

applications [24, 31, 174]. Another line of research focuses on revealing challenges and opportunities for safer smart contract development [119, 122, 188, 191].

The surveys most relevant to this work are concerned with the application of formal methods to modeling smart contracts. Singh et al. [156] analyzed 35 research works on formal verification and specification techniques as well as languages for smart contracts, together with the issues and vulnerabilities they address. Bartoletti et al. [32] compared five formal modeling techniques for Bitcoin smart contracts based on their expressiveness, usability, and suitability for verification. Furthermore, Ladleif and Weske [104] proposed a framework for evaluating formal modeling tools with respect to their capability to model a legal smart contract and assessed eight visual tools for smart contracts modeling using this framework. Different from the discussed works, our survey proposes a coherent conceptual framework that helps to establish the links between the approaches to formal modeling, specification, and verification. Our analysis is also more comprehensive in terms of the types of formal techniques covered and the application platforms considered.

There has been a body of literature dedicated to the collection of both bad and good ways of writing *secure* smart contracts. On one hand, the common pitfalls discovered in past experiences should be documented and avoided in the future development; and, on the other hand, the best practices that are proven to work can be summarized and adopted when facing a similar context. Atzei et al. [23] and some more recent work [79, 148, 182] collected common vulnerability patterns that might compromise the security of smart contracts. Permenev et al. [137] and Bernardi et al. [36] defined several classes of properties for which security analysts should watch out when performing contract audit. Several groups of researchers [31, 178, 181] proposed design patterns for smart contracts, which are general and reusable solutions for specific tasks. In this survey, we look beyond the security aspect and aim to provide a general taxonomy of domain-specific properties, which may also impact the correctness, privacy, efficiency, and fairness of smart contracts. We are also interested in establishing the links between the various types of properties and the formalism used in supporting them. We believe that this effort can help identify the gaps between the analysis needs and the existing technology capability.

We observe a consensus from the previous surveys that specifications for smart contracts are of paramount importance, but how to properly derive and represent them still remains an open challenge. This impacts not only the development of safe smart contracts, but also their verification and utilization [78, 92, 119, 174]. According to a recent survey on smart contract developers by Zou et al. [191], best practices, standards, and provably safe re-usable libraries are listed among the most desired improvements they would like to see happen in the smart contract ecosystem. While there are ongoing efforts towards this direction, such as the creation of curated datasets of smart contracts [62], we aim to facilitate the adoption of formal specification and verification by contributing the results of our analysis as well.

We adopt the same holistic view of verification techniques and tools for smart contracts. Still, our categorization of formal verification approaches is inspired by the previous studies on smart contract verification techniques performed by Harz et al. [85], Di Angelo et al. [59], and Miller et al. [122]. In this survey, we make the following novel contributions:

- A comprehensive survey of the state-of-the-art with broader scope;
- A four-layer framework for classifying smart contract analysis approaches;
- A taxonomy of smart contracts specifications in various domains;
- Discussions on observed trends and challenges in smart contract development and verification.

2 A TAXONOMY OF SMART CONTRACT MODELING FORMALISM

This section presents the formalisms used for formal modeling of smart contracts in the literature. The formalization approaches for smart contracts can be classified into two broad categories: *contract-level* and *program-level*, based on the level of abstraction at which modeling and analysis is performed. The *contract-level* approaches are concerned with the high-level behavior of a smart contract under analysis and usually do not consider technical details of its implementation and execution. These approaches usually describe the interactions between smart contracts and external agents mainly represented by users and the blockchain. The details of a smart contract are typically abstracted into a set of public functions that are invoked by users and observable results of their executions, such as emitted events or a change in a blockchain state. The *program-level* approaches mostly perform analysis on the contract implementation (i.e., source code or compiled bytecode) itself and thus are platform-dependent. By using program-level representations, such approaches enable precise reasoning about low-level details of the smart contract execution process.

2.1 Contract-level Models

In the contract-level analyses, smart contracts are viewed as black boxes that accept transaction messages coming from outside and possibly perform some computation based on them. This may result in externally observable events and even irreversible alteration to the blockchain state. The internal execution details and intermediate results are abstracted away in these types of models. Essentially, the contract-level models are concerned with the following concepts.

- *Users* – together with their balance, the transactions they initiate and the associated parameters such as the account addresses and the attached values. They can be any type of accounts that interact with the contract of interest—a user account or another contact. Some papers [25, 47, 106] also consider the choices of transactions made by users under different circumstances (i.e., strategies), as well as knowledge that is available to users at a particular moment of a smart contract execution [18, 25, 90, 169].
- *Contracts* – characterized by a set of publicly accessible functions and the externally visible effects of the function executions, e.g., changes of contract owners/balances, and emitted events/operations. As such, the behaviors of a contract are usually described by a set of *traces* [154], which are finite sequences of function invocations (i.e., the executed operations [121]), and properties are predicated on such traces.
- *Blockchain State* – including global variables referred to by the contracts and environment variables such as the timestamps and block numbers. More generally, blockchain state may also include the mining pool [10, 128] and the memory state of the contract execution environment [121].

The contract-level models are effective in expressing properties regarding the interactions between smart contracts and the external environment, and they are usually defined in terms of process algebras, state-transition systems, and set-based methods. The majority of such models are verified using techniques based on model checking [10, 15, 18, 29, 118, 121, 123, 128, 141]. At the same time, the abstraction over the execution logic of a smart contract introduces a gap between an original contract and its representation in a chosen formalism.

2.1.1 Process Algebras. Process algebra [28] is a family of mathematical approaches used to describe the behaviors of distributed or parallel systems in the form of interacting concurrent processes. The observable behaviors of a smart contract are similar to that of a shared-memory concurrent program [151] or a system with an interleaving mode [141], which makes process algebra a suitable candidate for high-level behavioral modeling. Therefore, process algebra captures

the interactions between concurrently acting users and one or several interconnected smart contracts [166]. For these interactions, correctness can be assured through a translation of a smart contract to one of the process-algebraic formalisms. An alternative approach involves implementation of a smart contract in a process-algebraic **domain-specific language (DSL)**.

Given that users interact with an Ethereum contract by calling its public functions, Qu et al. [141] and Li et al. [110] translate such functions from Solidity to process notations defined in ***Communicating Sequential Processes*** **(CSP)** and a variant of an applied π-calculus *SAPIC*, respectively. Figures 3 and 4 demonstrate the source code and a CSP model [141] of a **Safe Remote Purchase (SRP)** smart contract from the Solidity documentation [7]. The contract implements an escrow for mutually untrusting buyer and seller, who interact by invoking one of the functions of a smart contract. Figure 3 shows the Solidity implemetation of three functions, namely, `constructor()`, `abort()`, and `confirmPurchase()`. The model in Figure 4 represents each of these functions as a CSP process: `INIT`, `ABORT(x)`, and `ConfirmPurchased(x)`, respectively. Each process describes the corresponding actions of a smart contract and its users in terms of CSP *events*, such as payments made by participants and changes in a smart contract state. In case transactions are not atomic, the interplay between the functions invoked by different users may lead to risk conditions, which are manifested by vulnerable sequences appeared in the execution trace [141]. The authors of Reference [110] identify violations of trace-level safety properties that are commonly analyzed for token implementations, such as the immutability of the total number of tokens in a smart contract.

A process-algebraic model of a smart contract is adopted by several domain-specific programming languages [33, 142]. *Rholang* [142] is a Turing-complete language for the RChain platform, which supports concurrency and is based on the reflective higher-order process calculus ρ-calculus, a variant of π-calculus. Smart contracts in Rholang are processes that are triggered by messages from agents: users or other contracts. *BitML* [33] is a high-level smart-contract language for Bitcoin that is based on a process calculus with symbolic semantics. It is showcased in a series of publications by Bartoletti et al. [25, 33] how security properties, as well as arbitrary temporal properties, can be expressed and checked in BitML. To ensure correctness of a smart contract under different scenarios, BitML also supports specification of user strategies. Furthermore, smart contracts can also be soundly translated from BitML into transactions—a Bitcoin representation of a smart contract. The behavior of contract users can also be partially defined in a Turing-complete process calculus *scl* proposed by Laneve et al. [106]. scl models behaviors of a smart contract and a user as a parallel composition, where a user acts nondeterministically, while the behavior of a smart contract is determined by its execution logic. The scl models are in fact transition systems that are summarized by a first-order logic formula describing the values of the objective function for all possible runs and user strategies. Then, the game-theoretical analysis is performed to identify strategies that help users maximize their profits or minimize losses.

2.1.2 State-transition Systems. The behaviors of a smart contract can be naturally interpreted as a state-transition system, and in fact, Solidity encourages contracts being modeled as state machines [3]. Some next-generation smart contract languages, such as Obsidian [55] and Bamboo [2], follow a state-oriented programming paradigm that makes transitions between states explicit and therefore provides better security guarantees. When it comes to verification, depending on the properties interested, there is also a wide range of choices in modeling smart contracts as state-transition systems. For instance, timed automata, Markov decision processes, and Petri nets are used to capture the time, probabilistic, and multi-agent interactive properties of the systems, respectively. State-transition models are typically verified by model checkers against contract-specific properties, in addition to generic temporal logic properties such as

```
contract SafeRemotePurchase {
  enum State { Created, Locked, Release, Inactive }
  ...
  constructor() public payable {
    seller = msg.sender;
    value = msg.value / 2;
    require((2 * value) == msg.value);
  }
  // Abort the purchase and reclaim the ether.
  // Can only be called by the seller before
  // the contract is locked.
  function abort() public onlySeller
    inState(State.Created) {
    emit Aborted();
    state = State.Inactive;
    seller.transfer(address(this).balance);
  }
  // Confirm the purchase as buyer.
  function confirmPurchase() public
    inState(State.Created)
    condition(msg.value == (2 * value)) payable {
    emit PurchaseConfirmed();
    buyer = msg.sender;
    state = State.Locked;
  }
  ...
}
```

Fig. 3. Solidity source code of SRP contract [7].

$$
\begin{aligned}
&\mathbf{INIT} = init?msg\_value : Int \rightarrow \mathbf{if}(msg\_value\%2 == 0)\\
&\quad \mathbf{then}(seller.guaranty.eth.msg\_value \rightarrow\\
&\qquad state'!state = created \rightarrow PURCHASE(state))\\
&\quad \mathbf{else}(warning \rightarrow INIT)\\
&\mathbf{ABORT}(x) = access?object : Object \rightarrow \mathbf{if}(object == seller)\\
&\quad \mathbf{then}(state'?x \rightarrow \mathbf{if}(x == created)\\
&\qquad \mathbf{then}(abort \rightarrow state!inactive \rightarrow\\
&\qquad\quad eth.balance.seller.overall \rightarrow STOP)\\
&\qquad \mathbf{else}(warning \rightarrow STOP))\\
&\quad \mathbf{else}(warning \rightarrow STOP)\\
&\mathbf{ConfirmPurchased}(x) = access?object : Object\\
&\quad \rightarrow state'?x \rightarrow \mathbf{if}(x == created)\\
&\quad \mathbf{then}(buyer?value : Int \rightarrow \mathbf{if}(value == msg\_value)\\
&\qquad \mathbf{then}(purchaseConfirmed \rightarrow state'!locked \rightarrow STOP)\\
&\qquad \mathbf{else}(warning \rightarrow STOP))\\
&\quad \mathbf{else}(warning \rightarrow STOP)\\
&\ldots
\end{aligned}
$$

Fig. 4. A CSP model of SRP contract [141].

deadlock-/livelock-freedom (cf. Section 4.1). Section 3.2.1 provides a discussion on how the violation of these properties can lead to serious bugs. A variety of analysis frameworks already exist for the validation and verification of state-transition systems, e.g., BIP and cpntools, and so on. Such verification capacities combined with the ability to visually define a state-transition system make this formalism useful for smart contract synthesis [121, 192].

***Petri net* (PN)** is a mathematical language used to describe behaviors of concurrent systems. Unlike the aforementioned formalisms, it provides a graphical notation that portrays a system using a set of alternating places and transitions. PN tools allow a user to visually specify a smart contract model and simulate its possible behaviors. Similar capabilities of a ***Colored Petri net* (CPN)** extension and the supporting cpntools toolkit are utilized for security analysis of a Solidity contract [61, 115]. These two techniques construct CPN models of a contract and its users—manually [115] or automatically [61]—to identify if a vulnerable state of a net is reachable. The authors of Reference [61] make the simulation more realistic by additionally modeling the execution process of an EVM, considering gas consumption as well. The ability to visually define and execute a PN model also helps in synthesizing smart contracts from user inputs. A framework by Zupan et al. [192] generates a Solidity smart contract from a model defined as a PN workflow—a subtype of a PN describing business processes. Similar to the discussed PN approaches [61, 115], Zupan et al. automatically validate the model according to typical properties such as the absence of transitions that can never be executed [61, 115, 192] or reachability of a final state [192]. Through the visualization and validation of the model and its execution, a correct-by-design smart contract can be generated by a user without prior knowledge of smart-contract programming. The authors expect the technique to facilitate the application of smart contracts in use cases including supply chain [192] and legal contracts [104].

***Behavior-Interaction-Priority* (BIP)** is a layered framework used to model the interaction between smart contracts and users. BIP models of smart contracts usually consist of two layers: the components of the *Behavior* layer describe the logic of each smart contract as an FSM, while

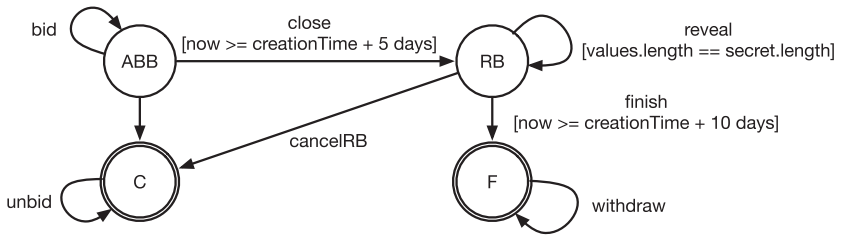

Fig. 5. A state-transition model of a Blinded Auction smart contract [121].

the *Interaction* layer defines the principles of their communication [15, 129]. Translation of a visually defined FSM model to BIP and, then, an input language of a NuSMV model checker, allows VeriSolid [121] to generate provably safe smart contracts. An example of a Blinded Auction smart contract [7] modeled as an FSM in VeriSolid [121] is shown in Figure 5. The model depicts possible transitions between four smart contract states: AcceptingBlindedBids (ABB), RevealingBids (RB), Finished (F), and Canceled (C).

***Timed automata* (TA)** capture the temporal behavior of a system. TA models allow measuring the impact of time on the execution of a smart contract [10] or reasoning about time constraints, which are essential in legal and commitment smart contracts [54]. Different from other approaches based on BIP [15, 121, 129], Abdellatif et al. [10] build TA models of not only a contract, but also its users, and a simplified blockchain mechanism, including mining of transactions in blocks. The interactions between these components are analyzed by statistical model checking to evaluate the chances of a successful attack on the contract. Andrychowicz et al. [18] use TA to describe behaviors of parties in a Bitcoin timed commitment. Commitment protocols require participants to exchange cryptographic hashes of the secret values they commit to before a stipulated time. To analyze correctness of this protocol, the authors additionally equip each TA with a structure describing the party's knowledge. The blockchain structure is maintained by a special agent, who is also modeled as an automaton.

A way to reason about the nondeterministic execution of a smart contract involves modeling it through probabilistic and multi-agent extensions of transition systems, such as *Markov decision processes* or *strategic* and *concurrent games* [40, 47, 106, 169]. To account for all possible user behaviors, Meyden [169] regards an atomic swap smart contract as a concurrent game. Both a model and its properties refer to the strategies of the contract participant. The authors apply the MCK model checker to establish correctness and fairness of a contract under the specified user behaviors. Chatterjee et al. [47] pursue a similar goal; however, their analysis of a concurrent game model is based on its objective function, similar to the *scl* process-algebraic technique [106]. To certify fairness and correctness of a smart contract, the authors of Reference [47] examine the expected payoff of its users and potential incentives for dishonest behavior. Different from the game-theoretical techniques, Madl et al. [118] capture the interaction between the users of a marketplace contract by describing their behaviors as *interface automata*. The authors verify a composition of the automata in a model checker to ensure that participants can successfully cooperate.

2.1.3 Set-based Methods. Set-based modeling frameworks *Event-B* and *TLA+* utilize set theory and logic to formally specify systems and have been applied on smart contracts as well. Analogically to other formal languages, such as PN, set-based frameworks support the tools for model analysis, making them useful in the implementation of correct-by-design smart contracts [30, 180] and verification of the existing ones [190]. The authors of References [30, 180, 190] utilize set-based formalisms to reason about safety properties of smart contracts. Although TLA+

supports temporal logic for specifying liveness, this feature has not yet been used to formalize smart contract requirements.

Models written in Event-B are discrete transition systems, which were demonstrated to be a suitable formal representation of a smart contract. Zhu et al. [190] automatically translate a contract written in a subset of Solidity to an Event-B model for refinement and verification via the `Rodin` platform. TLA+ is a formal specification language for concurrent systems. That allows Xu et al. [180] to model behaviors of interacting participants of a legal smart contract. Although the authors of Reference [180] do not translate their TLA+ specification to executable code, they propose a set of design patterns to protect them from common security vulnerabilities of smart contracts. In general, the set-based correctness requirements for smart contracts specify values allowed to be taken by contract variables [30, 190], possible states of contract participants [180], or the users that are authorized to invoke contract functions [190]. Xu et al. [180] also verify whether events representing the actions of contract parties are permitted according to the legal contract.

2.2 Program-level Models

The contract-level models are useful in reasoning about high-level interactions between the contracts and external parties, but do not help much in understanding the internal details about the contract execution. The latter is essential in establishing properties concerning the correctness and security of smart contracts. The program-level models aim to provide a white-box view of the target contracts based on lower-level representations, such as the source code, compiled bytecode, as well as derived analysis artifacts including AST, data- and control-flow graphs. However, program-level representations give intuition about the lower-level details of the execution process. The following are concepts that are relevant to this level of abstraction:

- ***Abstract Syntax Tree* (AST)** – representing smart contract source code (e.g., in Solidity) as hierarchical tree structures. It is often used to perform lightweight syntactic analyses on the contract implementations.
- *Bytecode* [137] (or *opcode*) – written in low-level machine instructions, which are closely tied to the execution environments (e.g., **Ethereum Virtual Machine (EVM)**). Since bytecode is obtained after compilation, it better reflects the machine-level specifics (e.g., gas consumption and exceptions), but at the same time may lose important source-level information.
- ***Control Flow Graph* (CFG)** – a graph representation of all program paths that might be traversed during execution. The CFG can be obtained from bytecode and is often used in static analyses of smart contracts, such as symbolic execution and automated verification.
- *Program Traces* – sequences of instructions (usually in bytecode) and events collected from executions at runtime. These traces reflect the exact behaviors of the contracts under specific inputs, which can be used as a source for performing dynamic analyses and runtime verification.

The program-level models preserve low-level execution details, and are therefore widely used in finding vulnerabilities and checking other security-related properties.

2.2.1 AST-level Analyses. Smart contract analyses performed directly on ASTs or similar parse-tree structures are often aimed at the checking of predefined code patterns [165, 182]. For example, Yamashita et al. [182] scan ASTs of Hyperledger Fabric [17] contracts written in Go to find dangerous code patterns and library imports. Another AST-based analysis is successfully applied in checking the compliance of a smart contract with respect to the ERC20 [171] standard by Chen et al. [49]. Moreover, AST can be used to facilitate lightweight pre-analyses: e.g., locating arithmetic operations that may cause overflows or assertions [170], decoding the memory layout information

to allow deeper inspection [42], and performing systematic instrumentation, such as vulnerability checks, to the contract code [12, 145]. Such pre-analyses can help improve the effectiveness and scalability of the more heavyweight downstream analyses. For instance, AST is the natural abstraction of smart contract code when it comes to statistical [112] and deep learning-based analyses [93].

The obvious drawback of these purely syntactic techniques is that they do not necessarily respect the operation semantics or the execution environment of smart contracts, thereby compromising the soundness/completeness of the analyses. For example, the gas consumption plays an important role in the functional correctness and security of Ethereum smart contracts, but is often neglected [165]. To allow for more sophisticated analyses, ASTs are often translated to derive other intermediate representations, such as CFG [12, 64], inheritance graph [64], and inter-contract call graphs [49].

2.2.2 Control-flow Automata. ***Control-flow graph*** **(CFG)** is often used to describe the operational semantics of a program. As the name suggests, CFG is a labeled directed graph where the graph nodes correspond to program locations and the edges correspond to possible transitions between the program locations. Thus, CFG can also be viewed as a type of state-transition system, except that it more closely reflects contract program executions than the contract-level state transitions discussed in Section 2.1. CFGs of smart contracts are mostly constructed from the compiled code—either EVM bytecode of Ethereum contracts [116] or **WebAssembly (WASM)** bytecode of EOSIO [4] contracts [87]. Still, recovering CFGs from low-level bytecode is non-trivial, at least in the case of Ethereum. There are several reasons for this. First, EVM is a stack-based machine, therefore, recovering the target address of a jumping instruction (destination of an edge) calls for context-sensitive static analysis capable of tracking the state of stack [45]. Furthermore, smart contract execution comprises a sequence of function invocations and, often, calls to other accounts, which demands inter-procedural and inter-contract analyses, respectively.

The types of analyses run on CFGs include control-/data-flow analyses and symbolic execution, which have matured tool supports for traditional software programs. Some of these tools and techniques are applied directly on the CFGs extracted from smart contracts, while others are custom built to accommodate particular language features of contract programs. As one example, `Ethainter` [43] performs information flow analysis to reveal composite vulnerabilities, i.e., reached only through a sequence of transactions, from traces of Solidity contracts. To reuse the existing program analysis tool-chain, namely, Datalog and Soufflé, the EVM bytecode is decompiled using `GigaHorse` [76] to recover the CFG with data- and control-flow dependencies. The same tool-chain is also used by `Securify` [168] and `Vandal` [44] in vulnerability detection after CFGs are constructed. In contrast, many custom symbolic execution engines have been built to traverse CFGs and check for interesting properties along the way. Some of the most notable ones include `Oyente` [116], `Mythril` [125], and `Manticore` [124]. These approaches are designed to identify vulnerable code patterns [116, 124, 125], integer arithmetic bugs [67], honeypots [167], gas-inefficient bytecode patterns [51], and so on.

To allow specification and verification of custom properties, several tools support extensions to CFGs with annotations. `Annotary` [177] extends `Mythril` [125] to also capture inter-contract and inter-transaction control-flows. `Solar` [108] allows annotations to a smart contract with constraints and invariants. `SmartScopy` [65] supports vulnerability patterns to be defined in their query language based on Racket.

2.2.3 Program Logics. To rigorously reason about the correctness of programs, various forms of program logic have been developed over the years, and some of them are successfully applied on smart contacts. A *program logic* is a proof system with a set of formal rules that allow to

statically reason about behavior of a program [21]. For example, Hoare logic [16], rewriting logic, and reachability logic [135] were used to prove correctness conditions (cf. Section 3.1.2) for smart contracts. An epistemic logic [90] describes the interactions between smart contract participants in terms of their knowledge, which facilitates verification of commitment and swap protocols (cf. Section 3.2.2). Correctness of smart contracts is also verified through a dynamic logic developed for Java programs [11, 34, 35]. Finally, formalisms that are based on deontic logic enable specification and verification of legal smart contracts, which are difficult to define formally otherwise [104].

Hoare and reachability logics assist reasoning about smart contract correctness by different formal techniques. Based on a formalization of EVM in *Lem* [89], Amani et al. [16] and Duo et al. [61] defined a sound Hoare-style program logic of Ethereum contracts. In verification of properties, however, the former approach relies on theorem proving, while the latter translates a formal model of a contract to CPN for model checking. In other approaches based on theorem proving, Hoare-style properties are verified for smart contracts written in Michelson [37]—a smart contract language of the Tezos [73] platform—or Lolisa [184]—a formal syntax and semantics of a Solidity language subset. Apart from the tools based on Hoare logic, $\mathbb{K}$ is a framework for design and development of language semantics based on rewriting logic [135]. $\mathbb{K}$ has been used to define a complete executable formal semantics of EVM [135] and Solidity [96], and a formal specification of an intermediate LLVM-like language for smart contracts IELE [98]. Properties that can be automatically proved by $\mathbb{K}$ include pre- and postconditions as well as reachability claims that are verified through matching logic. The expressiveness of $\mathbb{K}$ can be illustrated with a complete formalization of an ERC20 token contract [95]. Despite different choices of a verification technique, the aforementioned approaches are able to capture the dynamic aspects of smart contract execution, such as the memory state and gas consumption.

Such verification systems are usually built upon formal semantics of smart contract languages, e.g., EVM bytecode [16, 61, 89], Solidity [184], Michelson [37]. To make contracts amenable to formal reasoning, developers equip smart contract languages with a logical foundation, such as λ-calculus and formal semantics [46, 153].

At the same time, smart contracts that are written or translated to Java can be verified using a dynamic logic called JavaDL—a program logic for Java. A Java program annotated with JML specifications is automatically traslated to JavaDL by KeY—a framework for deductive verification. While Hyperledger Fabric contracts written in Java can be verified directly [34, 35], Solidity smart contracts are preliminarily translated to Java [11]. Dynamic logic can be seen as an extension of Hoare logic and is also used to define an executable specification for smart contracts [147]. The concepts related to dynamic logic are discussed in more detail in Section 3.1.1 and Section 3.1.2.

Some researchers consider logic-based languages to be particularly useful for the implementation and verification of *legal smart contracts* [75]. A legal smart contract is a digitized representation of a legal agreement that typically includes deontic modality statements describing obligation, permission, and prohibition of the parties. To capture contractual aspects, the authors of Reference [75] formalize a smart contract in a logic-based language called ***Formal Contract Logic* (FCL)**. Azzopardi et al. [27] propose a formal deontic logic with small-step operational semantics for Solidity smart contracts. Specifications defined in deontic and defeasible logics are discussed in Section 3.1.1.

3 SURVEY ON SMART CONTRACT SPECIFICATIONS

In this section, we review the literature on smart contract specifications. Requirements on smart contract applications are usually specified as *properties*—statements written in some formal languages about program behaviors. In Section 3.1, we survey the formalisms adopted in formal specification, which can be categorized into the contract- and program-level specifications. In

Section 3.2, we review various types of properties (e.g., security and functional correctness) from different application domains, observed in the published work under study.

3.1 Formal Specifications for Smart Contracts

Similar to the modeling formalisms discussed in Section 2, we make a distinction between *contract-level* and *program-level* formal specifications of smart contracts. This is in correspondence with the traditional classification of *model-oriented* and *property-oriented* specifications, rooted from *temporal logic* and *Hoare logic*, respectively. We discuss contract-level specifications about high-level contract properties in Section 3.1.1 and program-level specifications governing lower-level program implementations in Section 3.1.2.

3.1.1 Contract-level Specification. Contract-level specifications of smart contracts are expressed in various types of logic. The most prevalent category is a family of temporal logics, which includes linear and branching temporal logics and their dialects. Other logics include deontic and defeasible logics, which help to define the rights and obligations of smart contracts' parties in accordance with a legal contract. As specifications in temporal [121, 162], dynamic [147], and deontic logics [70] capture high-level behavioral characteristics of a system, they are used to synthesize correct-by-design smart contracts. Propositional and predicate logics usually convey factual statements about the contract's state over the traces.

Temporal Logics. Temporal logics express properties of a smart contract over time. The execution logic of smart contracts often encloses time constraints, be it legal contractual clauses [54] or ending time of an auction or voting contract [70]. As noted in Reference [128], in relation to smart contracts, the word "temporal" may refer to logical time considering only the order of events. Defined over a sequence of states, temporal formulae constitute a suitable language to describe the behavior of a state-transition system—a common formal representation of a smart contract described in Section 2.1.2, which is then verified by a model checker.

The two key temporal properties are *safety* and *liveness*. Safety properties correspond to a statement: "Nothing bad ever happens," and are often used to express invariance. In addition to general software properties such as deadlock-freedom, common safety properties of smart contracts define permissible values of state variables, establish invariants over a smart contract balance (cf. Section 3.2.3), specify access control for functions, or define a possible order of events.

Liveness properties state that "something good eventually happens." They characterize the possibility of a program to progress, which, for a smart contract, is often reflected in its ability to give away cryptocurrency. The primary liveness property is *liquidity* [33]—a security property discussed in detail in Section 3.2.1. Other liveness properties specify the ability of a smart contract or its users to eventually reach a particular desired state [118, 128], even considering different user strategies [169]. Unlike a safety property, however, there exists no finite counterexample for a liveness property. To make them amenable to symbolic analysis, some authors [131, 154] define under-approximations of liveness properties that are checked on traces of a fixed length.

***Linear Temporal Logic* (LTL)** is a fragment of **first-order logic (FOL)** that is used to define properties over a linear sequence of successive states of a program. LTL properties describe safety and liveness of transition-system models of smart contracts that are verified by a model checker [15, 25, 29, 33, 123]. Temporal safety properties of the Blinded Auction smart contract (Figure 5) in LTL [162] are depicted in Figure 6. ***Past-time LTL* (ptLTL)** has a set of temporal operators that reference the past states of a trace and allow succinct formulae for safety property [137]. For example, a ptLTL-based specification language for Ethereum [137] supports an operator *previously*, which refers to the value of a contract variable before the transaction execution. The properties defined

(i) `bid` cannot happen after `close`:
$$\mathbf{G}\ (\texttt{close} \rightarrow \neg \mathbf{F}\ \texttt{bid})$$
(ii) `withdraw` cannot happen before `finish`:
$$\texttt{finish}\ \mathbf{R}\ \neg\texttt{withdraw}$$

(i) `bid` cannot happen after `close`:
$$\mathbf{AG}\ (\texttt{close} \rightarrow \mathbf{AG}\ \neg\texttt{bid})$$
(ii) `withdraw` can happen only after `finish`:
$$\mathbf{A}\ [\neg\texttt{withdraw}\ \mathbf{W}\ \texttt{finish}]$$

Fig. 6. Properties of a Blinded Auction smart contract in LTL [162] (left) and CTL [121] (right).

in this language are then translated to source-code assertions. To account for the nondeterministic execution environment of a smart contract, Abdellatif et al. [10] specify properties in ***Probabilistic Bounded LTL (PB-LTL)***. Probabilistic properties in Reference [10] aim to estimate the probability of a successful attack on a name registration smart contract on each step of a transaction lifecycle.

***Computation Tree Logic* (CTL)** is a branching-time logic used to describe properties of a computation tree. In contrast to LTL formulae, whose semantics model considers one path (i.e., a sequence of states) without branching, CTL operators quantify over a tree of all possible future (branching) paths starting from the given state. Still, CTL is also used to formalize the specification of an FSM representation of a smart contract for model checking [118, 121, 128]. Figure 6 also illustrates a CTL formalization [121] of temporal properties for Blinded Auction (Figure 5). As in LTL, various dialects of CTL are used to formulate specific requirements of smart contract models. An extension of CTL for **Colored Petri Nets (CPN)**, *ASK-CTL*, is used to describe correctness of CPN models of smart contracts [61, 115]. To reason about the quantitative aspect of time in a timed commitment protocol, Andrychowicz et al. [18] use a ***Timed CTL* (TCTL)** language of the `UPPAAL` model checker to define properties of the timed automata models. ***Probabilistic CTL* (PCTL)** properties of a Markov Decision Process model of a selling contract measure the probability of one of the players to lose a certain percentage of his wealth after a given amount of time [40]. ***Alternating Temporal* (ATL)** and ***Strategic Logics* (SL)** are multi-agent extensions of CTL that allow quantification over agents' strategies. They describe the behavior of a concurrent game model of an atomic swap [169].

Dynamic Logic. Dynamic logic allows formulating properties in terms of *actions*, taking the termination of a program into consideration. The authors of Reference [147] propose a DSL for smart contract implementation based on properties written in ***Linear Dynamic Logic on finite traces (*LDL_f*)*.** Owing to its better expressiveness compared with LTL, LDL_f is used directly to model a smart contract, instead of encoding the state-transition model in a separate modeling language, such as Promela. Other applications of dynamic logic are related to deductive verification of smart contracts in the `KeY` framework. However, the properties are formulated in a smart contract as JML annotations [11, 34, 35] and are then translated to **Java Dynamic Logic (JavaDL)** (cf. Section 3.1.2).

Deontic and Defeasible Logics. Deontic logic is a division of modal logic operating over the concepts of obligation, permission, and prohibition. As legal prose relies on deontic modality [104], deontic-logic specification of a legal smart contract can bridge the gap between its legal semantics and implementation [54]. However, deontic modality requires special treatment from the specification perspective, as permission has no direct correspondence in many specification languages [27, 162]. To enable compliance checking between a legal contract and a Solidity smart contract, Azzopardi et al. [27] proposed an operational semantics for contracts written in deontic logic. Deontic statements also constitute one of the components of *ADICO institutional grammar* [57], which captures institutional rules, conventions, and norms. Given a set of ADICO rules, a framework proposed in Reference [70], can generate a skeleton of a Solidity smart contract.

Defeasible logic is a non-monotonic logic that allows for so-called defeasible propositions; in other words, exceptions to some of the rules. Such specifications contain a statement that

regulates superiority between the clauses of a contract, which helps to handle possible disputes and breaches. The authors of Reference [75] encode a smart legal contract that regulates a product license evaluation into a set of logical rules in ***Formal Contract Logic*** **(FCL)**—a formal contractual language based on defeasible logic and deontic logic of violations. The authors note that such logic-based implementation can be seen as an executable specification and, hence, facilitates monitoring.

Other Logics. We found several approaches that use other types of logic to describe contract-level requirements for smart contracts. For example, Li et al. [110] analyze traces of a process-algebraic contract model based on properties written in a fragment of (temporal) FOL. Temporal properties over traces of a *Scilla* smart contract are formalized using Coq higher-order logic predicates and a proposed temporal connective [152].

3.1.2 Program-level Specification. A dual type of smart contract specification is defined at the program level. In this section, we concentrate on Hoare-style specifications, which are often used together with program logics (cf. Section 2.2.3) or are instrumented in the actual smart contract source code. Then, we review specifications that rely on predefined patterns of instructions and events observed during smart contract execution.

Hoare-style Properties. A *Hoare-style* property of a program c is captured by a triple $\models \{P\}c\{Q\}$, where P and Q are predicates on the state of a program and its execution environment [16]. P and Q describe the state before and after the execution of c (if it terminates), respectively, and are referred to as a *precondition* and a *postcondition*. An *invariant* here is a predicate preserved by the execution of program c. Generally, Hoare triples express *partial correctness*, while *total correctness* requires proof of program termination. When applied to smart contracts, partial and total correctness are often considered equivalent due to the gas mechanism of Ethereum: A contract is guaranteed to terminate either successfully or due to an out-of-gas exception [16, 72].

A Hoare-style specification of a smart contract consists of preconditions, postconditions, and invariants defined for its functions, loops, and the contract in general. Predicates in these statements include storage and memory variables, environmental variables capturing block or transaction information, events and operations emitted during the execution. Hoare-style properties, therefore, can capture both vulnerable conditions and semantic correctness of smart contract functions. For example, access control in smart contracts is often realized via a precondition check on the identity of a transaction sender. As another example, postconditions are used to check if the constructor has initialized the contract state properly [176]. Hoare-style properties can be specified using program logics (cf. Section 2.2.3) and are often supported by verification languages, such as Why3 [127, 187]. Alternatively, Hoare-style specifications can be defined as smart contract assertions, which are analyzed by verification tools statically [12, 14, 83, 97, 137, 157, 176] or in runtime [107].

Many program logics listed in Section 2.2.3 enable Hoare-style reasoning about smart contracts in theorem provers [16, 37, 109, 184]. The work in Reference [16] demonstrates how this approach is used to describe correctness of an escrow smart contract. To consider all possible invocations, the authors also verify that any input that is not covered by a specification leads to a transaction being reverted.

Nehai et al. [127] define a Hoare-style specification of smart contract functions in the polymorphic FOL of `Why3`—an instrument for deductive verification. The specification contains requirements for both semantic correctness and lower-level operational details, such as gas consumption and the ordering of array elements. Other techniques based on deductive verification [11, 34, 35] use the `KeY` framework. In this case, the input consists of a Java smart contract annotated with ***Java Modeling Language*** **(JML)** properties expressing pre-/postconditions,

```
.decl uncheckedCall(u:Statement)

uncheckedCall(u) :-
  callResult(_, u),
  !checkedCallThrows(u),
  !checkedCallStateUpdate(u).
```

(a) Unchecked Send

```
.decl destroyable(stmt:Statement)

destroyable(stmt) :-
  !inaccessible(stmt),
  op(stmt, "SELFDESTRUCT").
```

(b) Accessible Selfdestruct

Fig. 7. Patterns for "Unchecked Send" and "Accessible Selfdestruct" vulnerabilities in Vandal [44].

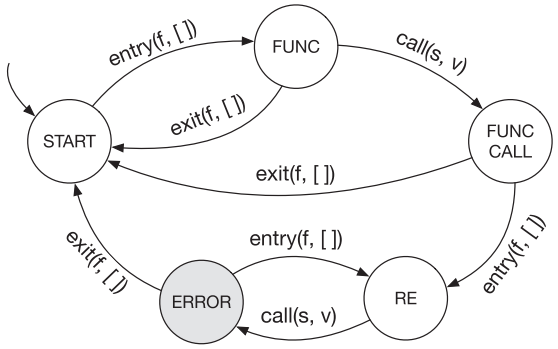

Fig. 8. An automata for identification of reentrancy during execution [111].

invariants, and modifiable variables of contract functions. The authors of Reference [35], however, note the inability of their approach to handle complex properties, including temporal, even if they are expressible in JML. JML specification is also supported by Raziel [146]—a framework for the implementation of *proof-carrying* smart contracts. Raziel supports JML for functional correctness annotations and *Obliv-Java*—for privacy variable annotations. A proof-carrying code proposal for safe smart contract upgrade by Dickerson et al. [60] also relies on a Hoare-triple specification.

Assertions that correspond to Hoare-style properties can be added to a smart contract by means of Solidity. The language includes `require` and `assert` statements that can express a precondition or an invariant and postcondition, respectively. `require` statements are widely used to validate the input received from a user—the absence of such checks is known to be a source of serious issues [79]. To advance further program-based specification, several tools provide custom specification languages [83, 107, 108, 137, 159, 176]. They support quantifiers [107], the implication operator [137], and auxiliary functions, such as *sum*, which refers to a sum of values stored in a mapping [83, 107, 137]. The latter is useful in specification of financial smart contract invariants, as shown in Section 3.2.3. To check semantic correctness of a smart contract, VeriSol [176] translates a state-machine policy of a Solidity smart contract into Hoare-style source code assertions. It can, therefore, identify an incorrect state transition and an incorrect initial state in a smart contract. Apart from user-defined specifications, several tools detect smart contract vulnerabilities by adding assertions to a smart contract automatically [12, 157]. Overall, instrumentation of a smart contract code with the corresponding checks introduces a transparent and consistent approach to smart contract specification. However, while intended to be side-effect free expressions, source-code assertions generate executable code and, therefore, gas consumption.

Program Path-level Patterns. Symbolic execution represents each program trace as a propositional formula over symbolic inputs to a program required to follow this execution path. Based on a contract CFG, this approach allows recognizing traces that contain problematic sequences of program states and instructions. In smart contract verification, the existing applications indicate if the contract is vulnerable [45, 67, 87, 116, 125, 131, 189], gas-inefficient [51], or a honeypot [167]. Some authors encode patterns over instructions and flow- and data-dependency in verification languages, such as *Datalog*, to certify contract security [44, 168] including the absence of gas-related vulnerabilities [77]. Figure 7 presents patterns that correspond to *Unchecked Send* and *Accessible Selfdestruct* [44] vulnerabilities. The topmost pattern detects a vulnerability if the

result of an external call is not used to update a blockchain state or throw an exception, while the bottom one detects a `selfdestruct` operation that can be called by an arbitrary account (cf. Section 3.2.1). The code snippets illustrating both vulnerabilities are shown in Figure 9.

While the pattern-based approaches discussed above operate on an EVM compilation of a smart contract, Perez and Livshits [136] use *Datalog* to formulate vulnerable patterns over execution traces retrieved from the blockchain. During fuzzing, `ReGuard` [111] compares an execution trace of a smart contract to an automaton that captures reentrancy. The automaton is shown in Figure 8, where `entry` and `exit` correspond to an entry and exit of a function, and `CALL` is the corresponding EVM instruction. Another common dynamic vulnerability detection technique identifies predefined patterns of instructions in runtime [71, 117].

Another type of path-level specification is based on events. Since the ERC20 standard specifies the events emitted by contract functions, several techniques perform compliance checking considering events, too [52, 83]. Events emitted by a smart contract during its execution are analyzed for compliance with business rules by Fournier et al. [68] and Molina-Jimenez et al. [123]. The former specification ensures acceptable transportation conditions in a supply chain use case, while the latter encodes clauses of a selling contract in terms of parties' responsibilities. Shishkin [154] formalizes functional requirements of an Ethereum contract as predicates over traces of events.

In addition, some publications adopt a different notion of events. For example, `EthRacer` [99] identifies event-ordering bugs in smart contracts by comparing results of different permutations of function invocations. Therefore, in this work, an event refers to a function call. The authors of Reference [80] verify serializability of smart contract execution by analyzing its traces of events, or transitions between contract states. The corresponding property of being ***effectively callback free*** **(ECF)** is satisfied if for all executions containing a callback there exists an equivalent execution without callback that can start in the same state and end in the same final state. A vulnerable pattern of events that correspond to actions performed by a smart contract helps to detect an attack in a CSP model of a Safe Remote Purchase smart contract [141] shown in Figure 4. The authors identify an attack through the existence of the following trace: $\langle init.msg\_value,\ seller.guarantee.eth.msg\_value,\ state'!created, \ldots,\ eth.balance.seller.overall,\ purchaseConfirmed,\ state'!locked,\ STOP\rangle$.

3.2 Properties Classification by Domains

Using the listed formalisms, specifications describe smart contracts' functional correctness from different perspectives. The considered aspects include absence of vulnerabilities, respect for privacy requirements, reasonable resource consumption, conformance to business-level rules, fairness to users. This section begins with an overview of existing classifications of domain-specific smart contract properties. Then, based on the property domains, we propose our categorization of smart contract properties that we have frequently seen in the research literature.

Existing Smart Contract Property Classification. Several taxonomies were proposed for smart contract vulnerabilities [23, 48, 79, 148, 182], with the first and, arguably, the most influential published in 2017 [23]. Section 3.2.1 overviews security properties that serve to protect a smart contract from types of vulnerabilities included in these classifications.

Two other publications describe common classes of domain-specific properties for smart contracts [36, 137]. These articles adopt dual forms of property specification: Hoare triples and temporal properties, respectively. Nevertheless, their categorizations share several types, such as *user-based access control* over contract functions and *invariants over aggregated values of collections.* In addition, Permenev et al. [137] define transition-system types of properties for smart contracts and extend invariants to a multi-contract setting. Some of these properties also resemble software

design patterns, which capture the recurrent problems in smart contract development, e.g., *State Machine*, *Access Restriction*, *Ownership* [178]. Bernardi et al. [36], however, consider two types of properties related to cryptocurrency exchange. We review some of these properties in Section 3.2.3.

Our domain-based analysis of properties is also shaped by a catalog of smart contract application domains proposed by Bartoletti et al. [31]. They divide smart contracts into *Financial, Notary, Game, Wallet, and Library* types. In Section 3.2, we describe our findings on the properties that are common for some of these types of smart contracts.

3.2.1 Security. Detection of security flaws in smart contracts received a lot of attention. Important security properties studied by the research community include *liquidity* ("a non-zero contract balance is always eventually transferred to some participants" [33, 131]), *atomicity* ("if one part of the transaction fails, then the entire transaction fails and the state is left unchanged" [116]), *single-entrancy* ("the contract cannot perform any more calls once it has been reentered" [149]), *independence of the mutable state and miner-controlled parameters* [78, 175]. Security guarantees of a smart contract also include proper *access control* [131, 168] for safety-critical operations, *arithmetic correctness* [65, 97, 157], and *reasonable resource consumption* [39, 51, 77]. The similarity of a smart contract to a concurrent object [151] suggests that some security guarantees can be derived from checking for *linearizability* [99] and *serializability*[80] of its executions.

The violation of these properties leads to many well-known smart contract vulnerabilities. For example, contracts violating liquidity are called "*greedy*," implying that their funds cannot be released under any conditions [131]. A smart contract violates atomicity if it does not check the return value of a (possibly failed) external call operation, which corresponds to the *unchecked call* vulnerability [78, 136]. The execution logic of a smart contract that is not independent of environmental variables, e.g., `block.timestamp`, is prone to *dependence manipulation* [175] by a miner, including the *timestamp dependency* vulnerability [116]. If the business logic of a smart contract depends on its mutable state parameters, such as balance and storage, then it has the ***transaction-ordering dependence* (TOD)** problem. TOD belongs to a class of ***event-ordering* (EO)** bugs, which arise from the nondeterministic execution of blockchain transactions [99] and are manifested by different outputs produced by different orderings of the same set of function invocations [99, 175]. The authors of Reference [99] use the notion of linearizability to identify EO bugs, including mishandled asynchronous callbacks from an off-chain oracle. The violation of single-entrancy implies that a smart contract is *reentrant*: provided with enough gas, an external callee can repeatedly call back into a smart contract within a single transaction. In execution traces, reentrancy is also identified as a violation of a serializability-like property [80], i.e., if the result of an execution with external callbacks is not equivalent to that of an execution without. Missing permission checks for the execution of a transfer or a `selfdestruct` operation make a smart contract *prodigal* and *suicidal* (Figure 9), respectively [131]. The identification of a transaction sender through the `tx.origin` variable is also considered vulnerable [44]. Integer arithmetics of EVM and absence of automatic checks for arithmetical correctness make Ethereum contracts prone to *integer overflow and underflow* [65, 157]. Figure 10 presents smart contract code containing the *integer underflow* vulnerability. Furthermore, the progress of a smart contract can be compromised by gas-exhaustive code patterns [51, 77].

Many symbolic execution tools detect vulnerabilities by scanning for vulnerable patterns in path conditions and traces derived from a contract CFG. For example, to identify *timestamp dependency*, `Oyente` [116] checks if a path condition of a trace includes the `timestamp` variable. To detect *suicidal* smart contracts, `sCompile` [45] checks whether, for the paths that contain `selfdestruct`, the path conditions constrain on the timestamp, block number, or the address of the contract owner. `Maian` [131] flags traces that transfer funds or control to an address not appearing in a contract

```
contract KotET {
  ...
  function withdraw(address to,uint amount) public {
    to.send(amount); //Unchecked Send
  }
  function close() public {
    selfdestruct(msg.sender); //Public Selfdestruct
  }
}
```

Fig. 9. "Unchecked Send" and "Accessible Selfdestruct" vulnerabilities [173].

```
contract UnderflowAttack {
  ...
  function withdraw (uint amount) public {
    require(balances[msg.sender] - amount > 0);
    msg.sender.transfer(amount);
    balances[msg.sender] -= amount; //Underflow
  }
  ...
}
```

Fig. 10. "Integer Underflow" vulnerability [81].

state, as *prodigal*. Scanning for vulnerable instruction patterns can also be done in runtime, for example, to identify that a permission check relies on `tx.origin` [50].

He et al. [86] consider prodigal and suicidal contracts to be the most challenging to detect, as it requires an ordered set of transactions. Indeed, if one sets the `owner` variable of a smart contract equal to his address, then he can bypass the aforementioned checks in the following transactions. The authors of Reference [43] call such vulnerabilities *composite* and detect them using information-flow patterns. Other static analysis tools also detect patterns over the data- and control-flow of a smart contract, capturing the necessary context-sensitive information. Figure 7(a) demonstrates a pattern for an *unchecked call* vulnerability detection used by `Vandal` [44]. The pattern states that the return value of the `call` operation should be used either to control the execution of the `throw` operation or to perform storage writes. To prevent *overconsumption of gas*, static analyses infer the upper bound on gas consumed by a function call [13, 39, 127] or detect vulnerable patterns, such as unreachable code [51] or an unbounded number of elements [77] in loops. Race conditions, such as TOD, are identified by static analyses as read-write hazards between the executions of two [78, 97, 168] or several [175] functions. An EO bug caused by an asynchronous callback could as well be prevented by a temporal property proposed in Reference [152]: It demands the state of a contract to remain unchanged until the response from an oracle is received.

As a liveness property, *liquidity* can be intuitively formulated in temporal logics [33]. Bartoletti et al. [33] additionally extend its definitions to user strategies: The default, *strategyless multiparty liquidity*, assumes the existence of a cooperative strategy to unfreeze funds. Liquidity can also be compromised by DoS attacks, for example, Parity Multisig Wallet [164] lost its ability to pay users as it delegated transfers to a smart contract that was destroyed. Nelaturu et al. [129] identify the Parity issue as a violation of deadlock-freedom.

Since arithmetic issues are often caused by non-validated function arguments, their identification is entrusted to pre- and postconditions [89, 98, 127, 135] and assertions [65, 157]. The (formally certified [149, 187]) `SafeMath` [133] Solidity library implements similar checks via the `require` statements. Source-code assertions are also a popular way to implement role-based *access control* in smart contracts. For example, it is done via equality checks between the values of `msg.sender` (the usage of `tx.origin` is considered vulnerable [44, 50]) and the corresponding contract variable, e.g., `owner` [83, 137, 176, 177]. Functions with restricted access usually change the value of a safety-critical variable, such as `owner`, or manipulate funds, e.g., transfer or mint tokens.

Reentrancy is one of the most studied smart contract vulnerabilities. Both static and dynamic data- and control-flow analyses detect the violation of *"no writes after call"* and similar patterns [65, 66, 108, 168] (yet, this pattern is neither sound nor complete approximation of reentrancy [149] and is considered a standalone property in Reference [168]). A similar check is also specified in terms of contract operations in CTL [121]. While *single-entrancy* is verified statically using Horn clauses in Reference [149], a similar property is also defined in terms of instructions in the execution trace [111] (Figure 8).

Finally, the inconsistency caused by reentrancy as well as other issues, such as flawed arithmetics [107], can be detected through invariants. Such invariants are proposed for financial [90, 110, 137, 154] (cf. Section 3.2.3) and voting contracts [130] (cf. Section 3.2.4).

3.2.2 Privacy. Smart contracts that implement auctions, games, and lotteries select a winner among the users who submitted their bids or moves. Blockchain transactions are publicly accessible, so a malicious user can decide on his move based on the actions of other contract participants. In a name registration smart contract, one can intercept the name chosen by another user from an unconfirmed transaction and register his address under this name first [10].

To prevent exposure of user inputs in a smart contract by design, it is recommended to implement cryptographic protocols, such as *commitment* and *timed commitment* [32] schemes. The protocol requires each participant to commit to a secret that corresponds to his input and should be revealed in time—otherwise, the participant has to pay compensation. In general, the correctness of a commitment smart contract is described in terms of the knowledge of each others' secrets that participants did or did not obtain as well as the corresponding changes in their balance. One example of such a property is proposed by the authors of Reference [18]: *"if an honest Bob did not learn Alice's secret then he gained Alice's deposit as a result of the execution."* Similar aspects are considered in specifications of *atomic-swap* contracts, which allow two parties to exchange their assets *atomically*, i.e., the swap either succeeds or fails for both parties [90, 169]. Alternatively, to prevent contract participants from cheating, Sergey et al. [152] suggested formulating a property as a knowledge argument over the prefix of observed execution history.

The transparency of the transactions and ledger data also makes smart contracts less applicable to privacy-sensitive domains, such as healthcare and voting. Potential solutions for privacy issues in smart contracts include the application of trusted management [101] or hardware [186]. On the whole, privacy-preserving smart contract is an independent topic of intense study, not covered by this survey. Among these approaches, we distinguish two that enforce privacy guarantees by allowing the corresponding annotations in smart contract code. To prevent data leaks, a smart contract language `zkay` [159] supports variable annotations, which specify who is eligible to read the value of a variable. Similar privacy annotations are supported by the `Raziel` [146] framework. As a demonstration, a smart contract is implemented in the Obliv-Java programming language with both privacy and correctness requirement annotations. Both tools utilize some zero-knowledge proof protocols to certify that privacy is maintained in a smart contract before it is executed.

3.2.3 Finance. One of the key advantages of smart contracts is their ability to manipulate digital assets, known as cryptocurrency. Smart contracts automate the execution of financial applications that involve storage and transfer of funds between users and contracts. Common examples include wallets, banks, escrows, cryptocurrency exchanges, and crowdfunding campaigns.

***Initial Coin Offering* (ICO)** is a method of raising funds by selling *tokens*—programmable assets managed by smart contracts on blockchain platforms [53]. Analysis performed by Chen et al. [53] demonstrates that 80% of ICOs are Ethereum-based smart contracts implemented according to the ERC20 standard [171]. The standard describes an interface of a token contract, i.e., its functions and events. Arguably, even this semi-formal specification gave rise to the application of formal techniques to token contracts [26, 52, 60, 84, 107–110, 137, 157, 161, 170] and development of a complete formal specification of ERC20 in $\mathbb{K}$ [95]. Several tools focus on consistency checking between a token implementation and a standard [52, 107, 108]. For example, `TokenScope` [52] identifies inconsistency in the histories of smart contract executions, while Li et al. [107] formulate a relevant set of invariants for several standards. Among other properties,

several more tools [50, 84, 170] check whether the contract emits the events determined by a specification, which can be expressed in a specification language proposed by Hajdu et al. [84].

Another common way to ensure validity of a token or other similar financial contract implementations is through correctness conditions for functions related to token transfers, e.g., `transfer()`, `transferFrom()`, and `approve()` in an ERC20 contract. For example, preconditions and postconditions of the `transfer()` function typically assert the following statements: *"the sender's balance is not less than the requested amount of tokens"* and *"the balances of both sender and receiver are updated according to the provided amount"* [11, 26, 97, 108, 109, 115, 135, 161, 170]. To cover the corner cases, some of the listed specifications explicitly state that *balances of non-participating users remain unchanged* [161]. Sun et al. [161] additionally check that *the value to be transferred is greater than zero*, a crowdfunding specified by Kalra et al. [97] only *accepts investments bigger than a threshold limit*, while Dickerson et al. [60] require *user balances to be non-negative*. The latter property can be found in specifications of banking contracts [11, 98], while wallets usually require the opposite: The authors of References [45, 97] request a user to *respect the limit* of Ether that can be transferred out of a contract within a transaction or a contract lifetime, respectively. A wallet specification by Kalra et al. [97] does not permit users to transfer funds to themselves, while ERC20-K [95] allows self-transfers but considers it a special case.

Formal specifications of financial contracts often contain vulnerability-related requirements, primarily integer overflow and underflow that may occur during a transfer [11, 83, 98, 157, 161, 170]. Other potential issues include event-ordering bugs: The authors of References [60, 99] show how an unforeseen ordering between the invocations of `approve()` and `transfer()` allows a malicious user to spend more funds than intended and how it can be prevented by an invariant. Wang et al. [175] detect an extended set of issues referred to as nondeterministic payment bugs, which include dependence manipulation conditions. Other specifications address access control [170, 177] and sending funds to invalid addresses [45, 50]. To assert that a crowdfunding contract can *pay back all investors if the campaign is not successful*, i.e., is not greedy, some specifications introduce temporal properties [137, 152]. To ensure that a smart contract has enough funds to do so, Permenev et al. [137] require that *the escrow's balance must be at least the sum of investor deposits unless the crowdsale is declared successful.* Other temporal properties ensure that funds are raised only until a cap [83] or a deadline is reached [137] or specify the order of states in a crowdfunding lifecycle and a corresponding payment procedure [137, 162].

An encouraged way to ensure the correctness of financial operations is by specification of invariants over a smart contract balance and an aggregation of user balances [83, 107, 110, 137, 161]. The most common invariant states that *the sum of user balances is equal to* `tokenSupply` [83, 107–109, 137, 154, 157, 161, 170], where the balances of users are stored in a so-called bookkeeping mapping variable, e.g., `balances`, and `tokenSupply` defines the total number of existing tokens. These invariants can also certify correctness of inter-contract communications of a smart contract [137]. For example, an invariant proposed by Wang et al. [173] ensures correctness for an account that receives funds from a smart contract: *"the amount deducted from a contract's bookkeeping balances is always deposited into the recipient's account."* To facilitate specification and verification of these invariants, several tools identify bookkeeping variables automatically [52, 172] or provide support for a special *sum* function, which can be applied to mappings and/or arrays [83, 137, 157]. Invariants are also used to express that the value of `tokenSupply` is immutable, bounded, or does not decrease [36, 109, 137, 170]. In addition, Bernardi et al. [36] consider invariants that apply to cryptocurrency exchanges, such as proportional distribution of tokens: *"one should not be able to exchange an asset worth nothing with a positively valued asset."*

Fairness in financial smart contracts can be analyzed by statistical means: The authors of Reference [40] estimate the probability of money losses for a participant of a goods exchange protocol. In a game-theoretical analysis of a crowdfunding smart contract, Chatterjee et al. [47] define the amount of sold tokens to be an objective function. If its value can exceed the number of available tokens, then the contract is declared unfair.

The authors of References [22, 105] define a set of properties for financial legal smart contracts, such as derivatives. The authors of *Marlowe* [105] verify that all smart contracts written in this language *conserve funds*, i.e., *the money that comes in plus the money in the contract before the transaction must be equal to the money that comes out plus the contract after the transaction, except in the case of an error.* A similar idea that money are neither created nor destroyed by a smart contract is addressed in Move [41] by using linear resource types for assets. In *Findel* [22], the requirements for financial derivatives include the following: *"derivatives should be free of calculation mistakes," "errors caused by external sources should be handled properly," "accidental swaps should be avoided (i.e., the generated transactions and contracts have the intended issuer and the intended owner)."*

3.2.4 Social Games. We associate the term *social game* with a class of smart contracts that regulate the participation and interaction of multiple users, according to some predefined game rules. Examples of such applications include auctions [20, 47, 121, 152, 162], voting schemes [20], lotteries [47], and games, such as gambling [11, 63, 97], rock-paper-scissors [34, 47, 152], and Ponzi-like games [121]. The payout to each participant is automatically determined by the contract source code, making the analysis of *fairness* even more relevant. Fairness of social games can be compromised due to privacy issues that occur if participants see each others' submissions on blockchains (cf. Section 3.2.2). In general, fairness specifications require user submissions to be valid and properly recorded, after which a correctly determined winner receives his payout. While its precise definition is subjective in the presence of several competing users [97], game-theoretical techniques [47, 106] analyze utility functions of participants as a reflection of contract fairness. The theory of mechanism design has also been applied in the verification of fairness properties of game contracts [113].

Social games typically require each user submission to be accompanied by a participation fee, be it a vote in a ballot [37], a bid in an auction [20, 47, 152], or a move in a game [97] (e.g., *"a minimum deposit is required to play the game"* [97]). In addition to that, users should have the permission to participate (*"people may vote if they are registered voters"* [70]), the choice submitted by a user should belong to a set of possible values (*"the vote goes for a registered candidate"* [37]) and, in some cases, *submitted only once* [97, 152]. Temporal properties are often used to assure that submissions are made before a deadline (*"people must not vote after the deadline"* [70]) or before the results are finalized ("`bid` *cannot happen after* `close`" [121]). Valid user inputs are to be correctly recorded by a smart contract [37, 152]. As an example, in an auction contract specified by Sergey et al. [152], the `pendingReturns` variable should store a mapping between each account and the sum of all his transfers made to a contract. A specification of a rock-paper-scissors contract by Sergey et al. [152] summarizes several of the listed requirements: *"each player can only submit their non-trivial choice once, and this choice will have to be a key from payoffMatrix in order to be recorded in the corresponding contract field."* Correctness of recording in voting contracts can also be guaranteed through the use of invariants over the number of votes [20] and individual user weights [107]. For instance, Antonino et al. [20] check that *the number of votes initially available is equal to the sum of votes still to be cast plus votes already cast throughout the election process.*

After user inputs are collected, it is the responsibility of a smart contract to determine the winner and perform the relevant transfers of cryptocurrency. The developers of smart contracts that

choose a winner randomly, such as lotteries [25] and casino [99], should be aware of the risks introduced by dependence on external oracles or randomization techniques, as discussed in Section 3.2.1. To enforce a correct winner selection in a rock-paper-scissors contract, Beckert et al. [34] encode the game rules in a postcondition of the corresponding function. The property of a simple Ponzi-scheme contract, King of the Ether Throne [1], certifies that *a user should always be crowned after successfully sending Ether to the contract* [121].

Upon the outcome finalization, most of the social game contracts transfer funds to some user, whether he is a successful gambler [63] or an unsuccessful auction participant [152]. In that sense, social games are similar to financial contracts with their properties and potential flaws. For example, the specification of an auction contract proposed by Sergey et al. [152] ensures that the smart contract does not send the funds arbitrarily and is able to pay out the unsuccessful participants: "*an account, which is not the higher bidder, should be able to retrieve the full amount of their bids from the contract, and do it exactly once.*" A similar liveness property is formulated by Shishkin [154] for a DAO-like smart contract, stating that *if some investor deposited some amount of money and did not vote for any proposal afterward, then he will always be able to get a refund by calling the corresponding function.* There are also specifications describing different ways to guarantee that a smart contract is not greedy and has enough funds to perform the required payments. For example, the owner of a casino in Reference [63] is only allowed to *withdraw from the pot when the bet of a player is resolved.* According to Beckert et al. [34], *as long as the auction remains open, the sum of the funds in the auction remains the same or increases but never decreases.*

Smart contract specifications can also describe the principal rules of its execution. For example, temporal properties that define the order of state transitions in an auction contract are shown in Figure 6. Another auction specification [35] states that *the items belong to the auctioneer as long as the auction is open and to the highest bidder after the auction is closed.* Invariants for an auction contract in Reference [113] require that *the bidder with the highest bid becomes the winner* and *the highest bid becomes the clear price.*

3.2.5 Asset Tracking. Supply chain is named to be one of the most suitable use cases for permissioned blockchains [68]. A supply chain typically involves several interacting parties responsible for production, transportation, and sales. Alqahtani et al. [15] describe the responsibilities of parties in a fuel supply chain, e.g., *"vendors must always maintain their production under the daily limit and log it into the ledger," "point-of-sales must always adhere to the price set by the regulators,"* and *"once the point-of-sales receives oil, a payment should eventually be triggered."* Another common requirement for supply chain contracts defines the acceptable transportation conditions, mainly, restrictions on the temperature [68] and humidity [176, 192]. The authors of References [176, 192] transfer the contract into a failed state if a critical reading is received, while Fournier et al. [68] report a violation if the amount of such readings exceeds a threshold within an hour.

The next group of asset-management smart contracts includes marketplaces, often the ones that facilitate trading between suppliers and producers of energy resources [121, 127, 128]. Similar to the properties of contracts from other domains, safety properties describe the correct execution flow of a market by declaring the allowed and required chains of actions [121, 128], e.g., *"if a user's payment has been received, then his bill is edited before the market closure."* Each marketplace contract implements an algorithm that matches the received buying and selling offers [121, 127, 128]. Nehai et al. [128] warrant a satisfactory solution by verifying that *the algorithm implements a repartition of energy proportional to payment* and that *once opened, the market will eventually be closed.* Properties from their later proposal [127] guarantee that the solution is optimal in the sense that it maximizes the total number of tokens exchanged.

A common example of legal smart contracts is a license agreement [74] for product evaluation. Its formal specifications [75, 162] describe the rights, obligations, and prohibitions of a user who participates in the evaluation. For example, one of the clauses states that *the Licensee must not publish comments on the evaluation of the Product, unless the Licensee is permitted to publish the results of the evaluation.* As discussed in Section 2.1, logic-based languages are practical for capturing the legal modality of such contracts [75]. Still, Suvorov et al. [162] encoded some of the clauses in temporal logic—although there is no direct representation of deontic modalities, the properties are still sufficient to identify a violation of certain contractual clauses. This approach also enables description of common principles of a contract operation, e.g., that *removal of a comment cannot happen if nothing has been published.* In general, formal representations of rights and obligations in smart legal contracts were discussed with regard to various formalisms including deontic logic [27], automata-based specifications [26], a domain-specific language based on LDL_f [147], set-based frameworks [30, 180], and some others. The authors of Reference [123] verify the formal contractual model of a data selling smart contract against typical contractual problems, such as clause duplication.

Several formal specifications exist for name registration smart contracts that manage domain names or provide aliases to blockchain account addresses [10, 88, 120]. Most of them focus on security analysis [10, 88], e.g., estimating the chance of a stealing attack [10]. On the contrary, Maksimov et al. [120] formulate a property regarding the functional correctness of a smart contract: *"the user can get an alias or a rejection after a number of attempts not exceeding a specified value."*

4 SMART CONTRACT FORMAL VERIFICATION TECHNIQUES

The selection of verification techniques largely depends on the type of a formal model and specification of a smart contract that one aims to analyze. This section discusses techniques, tools, and frameworks that are employed for verification of smart contract models described in Section 2. We also discuss the capabilities of these verification techniques with respect to the properties outlined in Section 3. A partial summary of verification techniques and employed tools is provided in Table 1.

4.1 Model Checking

Model checking is a well-established technique for automatically verifying a system model (with finite states) against its specification. When applied to smart contracts, model checkers perform verification of contract-level models, mainly transition systems, against a temporal logic specification [10, 15, 25, 40, 100, 118, 120, 121, 128, 169]. We identified 18 verification techniques that are based on model-checking, with seven utilizing the capabilities of `NuSMV` and `nuXmv` model checkers. The popularity of model checking is arguably caused by the suitability of both modeling and specification formalisms to smart contracts description combined with the existence of established automatic frameworks. Although model checking is successful in verifying systems of several smart contracts or users [15, 118, 129], its limitations are induced by the input language of a model checker and the state explosion problem [128].

Support of diverse transition systems and temporal properties (cf. Sections 2.1.2 and 3.1.1) help model checking capture different characteristics of smart contract execution, such as concurrency [61, 115, 141], nondeterminism [10, 40, 120, 169], or time constraints [18]. In addition to verifying the functional correctness of one contract [100, 121, 128], model checking handles systems of interacting smart contracts [15, 129] and users [118]. Furthermore, with specifications expressed in temporal logic, model checking is able to verify liveness properties and properties of progress, e.g., liquidity [33]. Model checkers, such as SPIN [91] or PAT [114, 160], additionally

Table 1. A (Partial) Summary of Formal Verification Tools

Verification Techniques	Tools	Selected References	Total
Model Checking	NuSMV, nuXmv	[15, 100, 118, 121, 128, 129]	18
	SPIN	[29, 123]	
	CPN	[61, 115]	
	BIP-SMC	[10, 120]	
	PRISM	[40]	
	UPPAAL	[18]	
	MCK	[169]	
	Maude	[25]	
	FDR	[141]	
	LDLf	[147]	
Theorem Proving	Coq	[19, 22, 37, 130, 132, 152, 161, 184]	20
	Isabelle/HOL	[16, 72, 88, 89, 105, 109]	
	Agda	[46]	
Program Verification	Datalog & Soufflé	[43, 44, 77, 136, 168]	42
	Boogie Verifier & Corral	[20, 83, 176]	
	LLVM & SMACK	[97, 175]	
	SeaHorn	[14, 97]	
	F*	[39, 78]	
	KeY	[11, 34, 35]	
	Why3	[58, 127, 187]	
	$\mathbb{K}$ framework	[96, 98, 135]	
	Custom static analyses	[64, 69, 105, 149, 153, 154, 157]	
Symbolic & Concolic Execution	`Oyente`	[67, 116, 167, 189]	22
	`Mythril`	[125, 140, 177]	
	`teEther`	[102]	
	`Maian`	[131]	
	`Manticore`	[124]	
Runtime Verification & Testing	`ContractLarva`	[26, 63]	26
	`EVM*`	[117]	
	`ReGuard`	[111]	
	`SODA`	[50]	
	`Solythesis`	[107]	
	`ECFChecker`	[80]	

verify conventional requirements of concurrent systems, such as deadlock- and livelock-freedom [29, 123].

However, model-checking suffers from the state explosion problem, which requires the users to apply abstraction techniques for smart contracts [25, 33] or assume a set of simplifications to its execution [100, 128]. Furthermore, Nehai et al. [128] claim that precise modeling of the blockchain environment is infeasible due to the limitations of the `NuSMV` model-checker input language. Indeed, as it checks a high-level representation, model-checking approaches rarely consider the details of smart contract execution on blockchain, such as the gas mechanism or a memory model. One of

the two identified exceptions includes a schematic model of blockchain behavior, such as mining of transactions in blocks [10]. A more realistic model of smart contract execution is achieved by Duo et al. [61] by combining CPN with a program logic for EVM bytecode.

4.2 Theorem Proving

Verification based on theorem proving involves encoding a system and its desired properties into a particular mathematical logic. Then, a theorem prover attempts to derive a formal proof of satisfaction of these properties based on the axioms and inference rules of the formal system. Unlike model checking, which only verifies finite-state systems, theorem proving supports verification of infinite systems [144]. However, theorem-proving techniques are usually semi-automated and require human involvement and expertise. Nevertheless, this drawback discourages neither academia nor industry from developing tools and languages that are backed by theorem proving. Our study identified 20 applications of theorem proving used to derive correctness guarantees for smart contract languages, individual smart contracts, and even verification frameworks.

Coq, Isabelle/HOL, and Agda theorem provers are used to develop formal semantics of low- [16, 37, 46, 72, 90, 132], intermediate- [38, 109, 152], and high-level [96, 183] programming languages for smart contracts, including DSLs for financial contracts [22, 105]. However, Li et al. [109] suggest that, among them, intermediate-level languages are the most suitable for formal verification. Annenkov et al. [19] note that the authors of formal semantics often focus on meta-theory, i.e., verification of language properties, as in case of *Plutus Core* [46] or *Simplicity* [132]. Established through its formal semantics, general properties of *Marlowe* [105] smart contracts include a bound on transactions that a contract can accept and preservation of money within the contract.

`ConCert` [19] is a Coq-based framework, which allows both meta-theoretic and functional reasoning about a (functional) language and a smart contract, respectively. Together with other publications [16, 19, 22, 37, 105, 109, 130, 184], it illustrates how theorem proving helps to precisely describe and prove correctness conditions of smart contract execution. These conditions include Hoare-style correctness properties over the state of a smart contract and its environment [16, 37, 109, 184], security requirements [22, 161], and gas consumption reasoning [72].

Despite the potential expressiveness of theorem-proving approaches, they seldom consider inter-contract communication and temporal properties of smart contracts. An attempt to examine smart contract interactions in Coq is performed by Nielsen and Spitters [130], who prove a voting contract invariant that approximates a temporal property. Verification of temporal properties, including liveness, in *Scilla* smart contracts, is enabled by a formalization of its trace semantics [152], while an embedding of the language in Coq is under development [153].

4.3 Symbolic Execution

Symbolic execution executes a program symbolically and thus is able to explore multiple concrete execution paths at a time. Symbolic execution of both Ethereum and EOSIO smart contracts is based on a traversal of a CFG reconstructed from the bytecode of a smart contract [87, 116]. Thereby, this approach inherits the difficulties associated with the issues of precise CFG modeling in smart contracts (cf. Section 2.2.2). Other difficulties encountered in symbolic execution for smart contracts include heavy usage of hash functions (hard to be solved by SMT solvers) and the need to symbolically model memory and smart contract interactions. Similar to model checking, symbolic execution approaches usually explore paths up to a certain length, which calls for the application of abstract interpretation and partial-order reduction. Nevertheless, symbolic execution is one of the principal approaches to detect smart contract vulnerabilities: Overall, we found 22 applications of symbolic and concolic executions. For solving path constraints, nine-tenths of these techniques rely on an off-the-shelf constraint solver `Z3`, however, Frank et al. [69] show that

other solvers, especially Yices2, significantly outperforms Z3. Symbolic execution is also used to guide smart fuzzing approaches [86].

In most cases, symbolic execution is used to detect predefined vulnerable patterns in the bytecode level. For example, that is performed by primary symbolic-execution tools that include Oyente [116], Mythril [125] and their extensions [67, 140, 167, 189]. EOSafe [87] verifies EOSIO smart contracts against a set of predefined vulnerabilities. teEther [102] examines execution paths of interest that contain a vulnerable instruction pattern and generates exploits to confirm the vulnerabilities. To extend verification capabilities beyond predefined patterns, SmartScopy [65] and Solar [108] provide support for user-defined vulnerable patterns and constraints, respectively.

Inter-transactional verification helps to uncover smart contract vulnerabilities that are composite, i.e., the vulnerability can only be exploited through a sequence of transactions [43]. However, symbolic execution suffers from the path explosion problem, which makes it unsound and unable to explore all possible sequences due to complex constraints. Therefore, most of the techniques, e.g., Maian [131], bound the invocation depth—the length of a considered sequence of transactions. VerX [137] combines symbolic execution with abstract interpretation and models gas mechanics of Ethereum, which allows elimination of the paths that are unreachable due to an out-of-gas exception. The tool checks reachability of assertions inserted in the smart contract source code from the temporal properties defined by a user. VerX also supports verification of a bundle of interacting smart contracts, which is rarely addressed by static analyses.

4.4 Program Verification

Smart contract correctness and security are verified by a wide range of program verification techniques, which total to 42 publications. The most numerous category among these techniques employs existing verification languages supported by verification frameworks, e.g., Boogie, LLVM, Datalog, or F*. Many of program verification tools are automated [64, 149, 157, 168] and can check for composite vulnerabilities [43] and semantic correctness of a smart contract [176]. The issues associated with program verification (especially static analysis) include the need to precisely abstract components of smart contract execution and memory model [20, 149], while the abstraction is often unsound [149]. Although static analysis tools often do not consider the gas mechanics of smart contract execution [20, 97, 149], others specifically focus on the estimation of gas consumption [13].

Translation of the smart contract source code into verification languages enables the wide capabilities of their toolchains. For example, data- and control flow analyses by Datalog and its Soufflé engine successfully detect vulnerabilities in bytecode [43, 44, 77, 168] and execution traces of Ethereum [136] smart contracts. Translation of Solidity—a high-level language—to Boogie enables verification of Hoare-style semantic properties [20, 176] and secure compilation of annotated smart contracts [83]. LLVM was shown to be a suitable IR for EVM [175], Solidity [97], and Go [97] implementations to check for both vulnerabilities and semantic conformance. Bhargavan et al. [39] performed embedding of both a Solidity subset and EVM bytecode in F* to allow verification of both high-level correctness and low-level properties, e.g., related to gas consumption. Translation of Solidity [127], EVM [187], and Michelson [58] smart contracts to WhyML enables deductive verification of Hoare-style properties. The authors of Reference [127] note that this technique facilitates the evaluation of a smart contract in a more realistic setting compared to model checking [128], which suffers from state explosion. However, unlike model checking [128], it does not allow specification of liveness properties. Deductive verification of smart contracts written or translated to Java w.r.t. Hoare-style properties is performed by the KeY framework [11, 34, 35]. A correct-by construction deductive verifier for EVM bytecode has been generated in the $\mathbb{K}$ framework [135].

Although several of the above-listed approaches [97, 168] make soundness claims, Schneidewind et al. [149] demonstrate sources of unsoundness identified in these analyses. The authors of Reference [149] propose a provably sound Horn-clause abstraction for EVM bytecode based on an EVM formalization [78] for reachability analysis, which enables verification of both security and functional Hoare-style properties. Other static analyzers support reasoning about execution traces [52] and smart contracts written in Scilla [153], Solidity [64, 154, 157] or EVM bytecode [149], and Tezla [143].

Some of the publications under study actively apply program verification tools that are specifically designed to operate over particular formalisms, such as set-based frameworks TLA+ [180] and Event-B [30, 190], a defeasible logic engine SPINdle [75] for logic-based programs, Tamarin prover for verification of concurrent protocols, C-language verifiers [14].

4.5 Runtime Verification and Testing

Runtime verification is a lightweight verification technique that checks the properties of a running program. Different from the techniques described in previous sections, runtime verification is concerned with one execution trace at a time. In the domain of smart contracts, the term *trace* often denotes a sequence of instructions executed by a blockchain platform, while it can also refer to a sequence of function invocations or events emitted by a smart contract. The availability of runtime information helps dynamic verification techniques mitigate one of the main obstacles to smart contract analysis—the need to model a complex execution environment of blockchain. Runtime verification usually provides a reactive defense against vulnerabilities or violations of correctness at runtime and can potentially identify vulnerable states that could not be reached by model checking or symbolic execution, due to the state or path explosion. Approaches based on runtime verification and testing, including fuzzing, account for 26 publications in our dataset.

Most of the runtime verification techniques perform verification on-chain, which allows them to reject vulnerable transactions acting as an online defense system. The approaches are based on enhancing the source code of a blockchain client [50, 66, 71, 80] or a smart contract itself [12, 26, 27, 107]. Both Go and JavaScript Ethereum clients enhanced with monitoring strategies and/or taint analysis are able to detect predefined vulnerable patterns in the instruction trace [71, 80, 117]. `Ægis` [66] and `SODA` [50] additionally support verification of user-defined patterns, which can, for example, indicate an inconsistency between a contract and a corresponding standard. Still, the discussion on the false positives produced by `SODA` [50] shows that although the tool can access runtime information, its completeness is still subject to the precision of the detection patterns. Another obstacle to wide adoption of such instrumented clients in Ethereum arises from the fact that it would require a hard fork of all EVM clients [107].

Another group of runtime approaches prevents vulnerabilities by inserting protection code into the source code of a smart contract [12, 26, 107]. For example, `Solythesis` [107] automatically instruments a Solidity smart contract with custom invariants. Compared to Solidity, its specification language for invariants contains additional features, including quantifiers and sums. Azzopardi et al. [27] instrument a smart contact with monitors that verify the compliance between a smart contract and a legal contract. However, the assertions generate additional gas consumption during the execution of an instrumented smart contract [107]. Another limitation of this approach is caused by the infeasibility to formulate and verify complex temporal properties, including liveness, using such smart contract instrumentation. While the above-listed tools perform on-chain verification, events emitted during the execution of a Solidity or a Hyperledger Fabric smart contract can also be checked for compliance off-chain [68, 123].

Besides runtime verification, smart contracts can be dynamically verified via fuzzing and testing techniques to identify and exploit vulnerabilities [111, 172]. To navigate a fuzzer towards higher

path coverage, some authors combine fuzzing with symbolic execution [86, 99]. Symbolic execution is also used to guide invariant-based testing of a smart contract by a federated society of bots [170].

5 DISCUSSIONS

In this section, we discuss our observations on the formal verification and specification literature of smart contracts to formulate answers to the research questions raised in Section 1.1. Some key observations are also summarized in Table 2.

5.1 Results

With **RQ1**, we aimed to identify formalisms that are currently used in the modeling and specification of smart contracts, as well as approaches to their verification. Through our study, we made the following observations. (1) Among contract-level representations, the dominant combination is state-transition models with specifications in temporal logic, which are then verified by model checkers. As seen from Table 2, this combination is mostly used to formulate functional correctness properties, and some security properties involving the concept of progress, such as liquidity [33], are modeled as liveness. (2) Another popular approach is automated program-level verification against Hoare-style specifications. In terms of functional correctness, program logics defined at the bytecode-level allow the reasoning about smart contract behaviors using theorem provers, while source-level annotations are usually discharged by program verification techniques. Verification of security properties is dominated by existing well-established symbolic execution and program verification tool-chains, e.g., Boogie. With a few exceptions, these approaches identify security issues along program paths that match predefined vulnerable patterns. (3) Runtime verification checks for similar issues but on runtime execution traces. These techniques often require instrumentation of smart contract code with proper monitors, e.g., assertions. Runtime monitoring can also be implemented off-chain with tracking of events for compliance checking for escrow and supply chain applications [68, 123].

5.2 Open Challenges

With **RQ2** and **RQ3**, we aimed to identify current challenges and limitations in formal verification of smart contracts, respectively.

Pattern-based Verification. As demonstrated in Table 2, smart contract security analysis is mostly performed at the program level. These techniques typically rely on *vulnerable patterns* that were defined by experts, which limits these tools to the identification of a known set of vulnerabilities [175]. The generalization to larger classes of vulnerabilities can be achieved by focusing on the root causes of bugs, such as the influence of nondeterministic factors on smart contract execution [175]. To expand the list of detectable vulnerabilities, some works propose specification languages that allow users to define vulnerabilities [65], temporal properties [137], or annotate Solidity contracts with requirements, in forms of invariants [83, 173].

Limited Verification of Temporal Properties. Temporal logic is used as a universal formalism for high-level correctness specification for smart contracts. In nearly all cases, temporal properties are specified over a contract-level model of a smart contract, which typically abstract technical details of its operation. At the same time, program-level techniques that provide better precision in terms of the execution environment modeling rarely support specification and verification of temporal properties. Nevertheless, the wide use of temporal properties in smart contract specification suggests that they should be analyzed to the precision of program-level techniques.

Table 2. A (Partial) Overview of the Formalization and Verification Literature

Domains	Applications	Model Formalisms				Specification Formalisms				Verification Techniques				
		Process Algebra	Transition System	Control-Flow Automata	Program Logic	Temporal Logics	Other Logics	Hoare Logic	Path-level Patterns	Model Checking	Theorem Proving	Symbolic Execution	Program Verification	Runtime Verification
Finance	ICO / Token	[110]Ξ	[121]Ξ	[137]Ξ	[135]Ξ	[121, 137]Ξ	[135]Ξ [110]Ξ	[107, 109]Ξ	[52]Ξ	[121]Ξ	[109]Ξ	[137]Ξ	[110, 135]Ξ	[52, 107]Ξ
	Bank		[61, 100]Ξ		[61]Ξ	[61, 100]Ξ		[11]Ξ	[39]Ξ	[61, 100]Ξ			[11, 39]Ξ	
	Wallet		[129]Ξ		[37]❖	[129]Ξ		[37, 58]❖		[129]Ξ	[37]❖		[58]❖	
	Escrow / Purchase	[141]Ξ	[61]Ξ		[16, 27, 61]Ξ	[61]Ξ		[16]Ξ	[27, 141]Ξ	[61, 141]Ξ	[16]Ξ			[27]Ξ
Social Games	Auction		[121]Ξ			[121]Ξ		[187]Ξ [35]★		[121]Ξ		[113]Ξ	[113, 187]Ξ [35]★	
	Voting		[61]Ξ		[61]Ξ	[61]Ξ		[37]❖ [20, 107]Ξ		[61]Ξ	[37]❖	[113]Ξ	[20, 113]Ξ	[107]Ξ
	Games / Gambling		[162]Ξ			[162]Ξ		[34]★	[63]Ξ				[34]★	[63]Ξ
Asset Tracking	Supply Chain		[15]★			[15]★		[176]Ξ	[68]★	[15]★			[176]Ξ	[68]★
	Marketplace		[128]Ξ			[128]Ξ		[127]Ξ		[128]Ξ			[127]Ξ	
	License Agreement		[162]Ξ		[75]✚	[162]Ξ	[75]✚						[75]✚	
	Name Registration		[10]Ξ			[10]Ξ	[88]Ξ			[10]Ξ	[88]Ξ			
Protocols	Timed Commitment	[25]₿	[18]₿			[18, 25]₿				[18, 25]₿				
	Atomic Swap		[169]✚		[90]✚	[169]✚	[90]✚			[169]✚	[90]✚			
Security	Reentrancy		[121]Ξ	[116]Ξ		[121]Ξ	[89]Ξ [130]✚		[111, 116, 175]Ξ	[121]Ξ	[89]Ξ [130]✚	[116]Ξ	[175]Ξ	[111]Ξ
	Concurrency	[141]Ξ		[99]Ξ					[99, 141, 175]Ξ	[141]Ξ		[99]Ξ	[175]Ξ	
	Dependence Manipulation			[116]Ξ [87]†					[116, 117, 175]Ξ [87]†			[116]Ξ [87]†	[175]Ξ	[117]Ξ
	Unchecked Call			[116]Ξ					[50, 116, 175]Ξ			[116]Ξ	[175]Ξ	[50]Ξ
	Access Control	[110]Ξ		[137]Ξ [87]†		[137]Ξ	[110, 161]Ξ	[176]Ξ	[43, 50]Ξ		[161]Ξ	[137]Ξ [87]†	[43, 110, 137, 176]Ξ	[50]Ξ
	Liquidity	[33]₿	[121]Ξ	[131, 168]Ξ		[33]₿ [121]Ξ	[152]✚		[131, 168]Ξ	[33]₿ [121]Ξ	[152]✚	[131]Ξ	[168]Ξ	
	Resource Consumption			[51, 77]Ξ			[72]Ξ	[127]Ξ	[51, 77]Ξ		[72]Ξ	[51]Ξ	[77, 127]Ξ	
	Arithmetic			[67]Ξ			[161]Ξ	[157]Ξ	[67, 117]Ξ		[161]Ξ	[67]Ξ	[157]Ξ	[117]Ξ

Ξ: Ethereum, ₿: Bitcoin, ★: Hyperledger Fabric, ❖: Tezos, †: EOS, ✚: Other.

Complications in Execution Environment Modeling. Arguably, the major difficulty of all techniques that perform static analyses lies in the modeling of intricate aspects of smart contract execution, which also differ across platforms. For example, a verifier for Ethereum smart contracts should precisely model the gas mechanism [137] and the memory model [20, 69] with its hash-based object allocation [137]. Gas consumption in Ethereum is determined by the executed instructions at runtime, so gas-aware analyses have to be carried out at the bytecode-level, which makes specification of high-level properties less straightforward [20]. Precise reasoning about temporal properties concerned with physical time is also complicated by the absence of a global clock [47] and the delay in transaction processing [122]. Execution of Ethereum contracts also introduces many sources

of nondeterminism, including transaction ordering and the fallback mechanism [175], only determined at runtime. Still, comprehensive verification of functional correctness and security of smart contracts should allow for inter-contractual reasoning [102], which accounts for a nondeterministic behavior of an external callee. In some platforms [17], nondeterminism can also arise from the development of smart contracts in general-purpose languages, which can demonstrate nondeterministic behavior. However, to ensure consensus checking between blockchain nodes, the usage of nondeterministic methods is strongly discouraged and can be detected statically [182] or in combination with dynamic analysis [158]. Other aspects of blockchain that require modeling include the possibility of forks [24, 90] and reverted transactions.

Limitations of Runtime Verification. Since smart contracts are immutable once deployed, the discussed techniques are mostly to be applied in design-time. Even in the domain of runtime verification, many techniques [26, 27, 107] rely on the instrumentation of contracts with protecting code, which has to be performed pre-deployment as well. Moreover, instrumentation introduces additional gas consumption [107] and increases the size of bytecode, which is limited [50]. Runtime verification of smart contracts is also performed through the application of instrumented Ethereum clients [50, 71, 80, 117]. They, however, can only operate locally—otherwise, a hard fork is required for its integration into a platform [107]. In case the information about the execution is retrieved from the blockchain, information about private contract variables may not be observable [170]. Furthermore, if transactions are analyzed offline, then suspicious transactions cannot be reverted in runtime [50].

Fragmented Standards and Ecosystems. Despite the extensive literature, there is still not a clear path towards safe smart contract development [191]. This is particularly reflected by the lack of standard development tool-chains and the difficulty in choosing the right techniques. Arguably, the difference between the chosen formal specification and verification techniques as well as different threat models are the key reasons for a low agreement between smart contract verification tools demonstrated by recent studies [136, 175]. In addition, our observations coincide with the opinion of He et al. [87] that smart contracts operating on different platforms require substantially different approaches to specification and vulnerability detection. Although Table 2 demonstrates that a vast majority of approaches concentrate on verification of Ethereum smart contracts, our study includes the work on Bitcoin [18, 25], Tezos [37, 38, 143], Hyperledger Fabric [15, 35, 68, 97, 118, 147, 182], and EOS [87].

5.3 Future Directions

We envision a few promising research directions in the area of formal specification and verification of smart contracts.

Safe Languages. One of the emerging directions is to develop *safe smart contract languages* that eliminate many security risks by design or facilitate program verification. For example, Tezos supports three formally certified programming languages varying in platform abstraction level [37, 38, 143], shipped with associated verification frameworks. Besides formal semantics, other guarantees are derived from linear resource types in Move [41] and Flint [150], explicit state changes of Bamboo [2] and Obsidian [2, 55], functional programming principles and a restricted instruction set in Scilla [153], user-centered design as in Obsidian [55], and so on. To allow automated formal verification, some languages (e.g., DAML [8], Pact [138], and TEAL [9]) are also intentionally designed as non-Turing complete. We also noticed an increasing number of *domain-specific languages*, for example, Marlowe [105], which should simplify the development and verification of financial smart contracts.

Self-healing Contracts. Another prominent direction is concerned with recovering from runtime failures/violations, a.k.a. *automated repair* of contracts. In 2017, Magazzeni et al. [119] suggested that AI planning techniques can be used to automatically patch the misaligned traces of smart contracts. One of the recent examples is an automated gas-aware repair technique for Ethereum smart contracts proposed by Yu et al. [185], which is based on a genetic mutation technique guided towards lower gas consumption. To ensure validity of a generated patch w.r.t. the legitimate behavior of a vulnerable smart contract, the authors automatically construct a test suite based on its previous blockchain transactions. The generated patches should, therefore, fix the identified vulnerability without breaking any of these previously passing tests. The gas constraints of the Ethereum blockchain introduce an additional criterion for the validity of a patch, which should minimize the gas consumption. The ability of a smart contract to recover from a violation could also be relevant in the context of smart legal contracts, e.g., to resolve disputes and conflicts between participants of a contract [56].

Hybrid Analyses. In the verification of smart contacts, better results can be achieved through a combination of online and offline techniques [50, 107]. A combination of contract-level and program-level approaches to smart contract modeling can additionally help mitigate the trade-off between the high-level behavioral specification and the precise modeling of execution details. Filling in the gap between high- and low-level analyses can enable accurate verification of interesting temporal properties. The study of smart contract fairness is complicated by the multi-user environment of its execution model and unknown intentions of contract designers. Still, to analyze such high-level properties, including fairness, researchers currently apply techniques from the domains of game theory and mechanism design [47, 106, 113].

Collaborative Development of Standards. Some work has been done to promote standards and best practices in the development of smart contracts, for example, collections of design and security [178, 181] patterns. Two recent publications provided taxonomies of common smart contract properties and invariants [36, 137], while the authors of Reference [79] summarized their observations on the performance of static and dynamic analysis tools w.r.t. different security issues. The adoption of formally verified libraries is another step towards safer smart contract development. Two recent works [149, 187] used formal verification to assure the safety of the popular `SafeMath` library for Solidity. Other contributions include creation of a benchmark and curated datasets of vulnerable smart contracts [62] as well as evaluation frameworks for static analyses based on vulnerable code injection [12]. The development of standard benchmark is facilitating advances in the ML/DL approaches to smart contract analysis—another trend in smart contract analysis that we observed in recent publications. We expect the work in this direction to continue.

6 CONCLUSION

In view of the increasing adoption of blockchain and smart contract technologies, formal specification and verification of smart contracts has been an active topic of research in recent years. Within this survey, we have presented a comprehensive analysis and classification of existing approaches to formal modeling, specification, and verification of smart contracts. Furthermore, we outlined common properties from different smart contract domains and correlated them with the capabilities of existing verification techniques. Our findings suggest that a combination of contract-level models and specifications with model checking is widely used to reason about the functional correctness of smart contracts from all the considered domains. However, program-level representations are often analyzed from the perspective of security properties, which are checked by means of symbolic execution, program verification toolchains, or theorem proving. Security issues can also be identified in execution traces of smart contracts by runtime verification techniques. A

holistic perspective on smart contract analysis, which was adopted in this survey, allowed us to uncover the existing limitations to effective formal verification of smart contracts and suggest future research directions that may be taken to overcome them. We envision further advancement in the areas including safe smart contract language design, automated repair of smart contracts, hybrid contract- and program-level approaches to formal modeling and specification of smart contracts, and collaborative development of standards for safe smart contract implementation.

REFERENCES

[1] 2016. King of the Ether Throne — Post-Mortem Investigation. Retrieved from https://www.kingoftheether.com/postmortem.html.

[2] 2018. Bamboo: A Morphing Smart Contract Language. Retrieved from https://github.com/cornellblockchain/bamboo.

[3] 2020. Common Patterns — Solidity 0.6.11 documentation. Retrieved from https://solidity.readthedocs.io/en/v0.6.11/common-patterns.html.

[4] 2020. EOS.IO Technical White Paper v2. Retrieved from https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[5] Etherscan. 2020. Ethereum Charts and Statistics | Etherscan. Retrieved February 13, 2020 from https://etherscan.io/charts.

[6] 2020. Solidity — Solidity 0.6.11 documentation. Retrieved from https://solidity.readthedocs.io/en/v0.6.11/.

[7] 2020. Solidity by Example — Solidity 0.6.11 documentation. Retrieved from https://solidity.readthedocs.io/en/v0.6.11/solidity-by-example.html.

[8] 2021. Daml Programming Language. Retrieved from https://daml.com.

[9] 2021. Transaction execution approval language (TEAL) specification. Retrieved from https://developer.algorand.org/docs/reference/teal/specification.

[10] T. Abdellatif and K. L. Brousmiche. 2018. Formal verification of smart contracts based on users and blockchain behaviors models. In *Proceedings of the IFIP NTMS*. IEEE, 1–5.

[11] W. Ahrendt, R. Bubel, J. Ellul, G. J. Pace, R. Pardo, V. Rebiscoul, and G. Schneider. 2019. Verification of smart contract business logic. In *Proceedings of the FSEN*. Springer International Publishing, Cham, 228–243.

[12] S. Akca, A. Rajan, and C. Peng. 2019. SolAnalyser: A framework for analysing and testing smart contracts. In *Proceedings of the APSEC*. 482–489.

[13] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. 2019. GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. arXiv:1912.11929.

[14] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. 2019. SAFEVM: A safety verifier for ethereum smart contracts. *CoRR* abs/1906.04984 (2019).

[15] S. Alqahtani, X. He, R. Gamble, and P. Mauricio. 2020. Formal verification of functional requirements for smart contract compositions in supply chain management systems. In *Proceedings of the HICSS*.

[16] S. Amani, M. Bégel, M. Bortin, and M. Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the ACM CPP*. ACM Press, 66–77.

[17] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, and Y. Manevich. 2018. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the EuroSys*. ACM.

[18] M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. 2014. Modeling bitcoin contracts by timed automata. In *Proceedings of the FORMATS*. Springer International Publishing, 7–22.

[19] D. Annenkov, J. B. Nielsen, and B. Spitters. 2020. ConCert: A smart contract certification framework in coq. In *Proceedings of the ACM CCP*. ACM, 215—228.

[20] P. Antonino and A. W. Roscoe. 2020. Formalising and verifying smart contracts with Solidifier: A bounded model checker for Solidity. arXiv:2002.02710.

[21] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press.

[22] A. Arusoaie. 2020. Certifying Findel Derivatives for Blockchain. arXiv:2005.13602.

[23] N. Atzei, M. Bartoletti, and T. Cimoli. 2017. A survey of attacks on ethereum smart contracts. In *Proceedings of the POST*. Springer, 164–186.

[24] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. 2018. SoK: Unraveling bitcoin smart contracts. In *Proceedings of the POST*. Springer International Publishing, 217–242.

[25] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino. 2019. Developing secure bitcoin contracts with BitML. In *Proceedings of the ACM ESEC/FSE*. ACM, 1124–1128.

[26] S. Azzopardi, J. Ellul, and G. J. Pace. 2019. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Proceedings of the RV*, Vol. 11237. Springer Verlag, 113–137.
[27] S. Azzopardi, G. J. Pace, and F. Schapachnik. 2018. On observing contracts: Deontic contracts meet smart contracts. In *Proceedings of the JURIX*, Vol. 313. IOS Press, 21–30.
[28] J. C. M. Baeten. 2005. A brief history of process algebra. *Theor. Comput. Sci.* 335, 2 (2005), 131–146.
[29] X. Bai, Z. Cheng, Z. Duan, and K. Hu. 2018. Formal modeling and verification of smart contracts. In *Proceedings of the ACM ICSCA*. ACM Press, 322–326.
[30] R. Banach. 2020. Verification-led smart contracts. In *Proceedings of the FC*. Springer International Publishing, 106–121.
[31] M. Bartoletti and L. Pompianu. 2017. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Proceedings of the FC*, Vol. 10323 LNCS. Springer Verlag, 494–509. arXiv:1703.06322.
[32] M. Bartoletti and R. Zunino. 2019. Formal models of bitcoin contracts: A survey. *Front. Blockch.* 2 (2019), 8. Retrieved from https://www.frontiersin.org/article/10.3389/fbloc.2019.00008.
[33] M. Bartoletti and R. Zunino. 2019. Verifying liquidity of bitcoin contracts. In *Proceedings of the POST*. Springer International Publishing, 222–247.
[34] B. Beckert, M. Herda, M. Kirsten, and J. Schiffl. 2018. Formal specification and verification of hyperledger fabric chaincode. In *Proceedings of the SDLT*.
[35] B. Beckert, J. Schiffl, and M. Ulbrich. 2019. Smart Contracts: Application Scenarios for Program Verification. Retrieved from https://www.key-project.org/wp-content/uploads/2019/11/sc-verification.pdf.
[36] T. Bernardi, N. Dor, A. Fedotov, S. Grossman, N. Immerman, D. Jackson, A. Nutz, L. Oppenheim, O. Pistiner, N. Rinetzky, M. Sagiv, M. Taube, J. A. Toman, and J. R. Wilcox. 2020. WIP: Finding bugs automatically in smart contracts with parameterized invariants. Retrieved from https://www.certora.com/pubs/sbc2020.pdf.
[37] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. 2019. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. arXiv:1909.08671.
[38] B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson. 2020. Albert, an Intermediate Smart-Contract Language for the Tezos Blockchain. arXiv:2001.02630.
[39] K. Bhargavan, N. Swamy, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, and T. Sibut-Pinote. 2016. Formal verification of smart contracts. In *Proceedings of the ACM PLAS*. ACM Press, 91–96.
[40] G. Bigi, A. Bracciali, G. Meacci, and E. Tuosto. 2015. *Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods*. Springer International Publishing, 142–161.
[41] S. Blackshear, D. L. Dill, S. Qadeer, C. W. Barrett, J. C. Mitchell, O. Padon, and Y. Zohar. 2020. Resources: A Safe Language Abstraction for Money. arXiv:2004.05106.
[42] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. 2018. SmartInspect: Solidity smart contract inspector. In *Proceedings of the IEEE IWBOSE*. 9–18.
[43] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the ACM PLDI*. ACM, 454–469.
[44] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981.
[45] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang. 2019. sCompile: Critical path identification and analysis for smart contracts. In *Proceedings of the ICFEM*. Springer International Publishing, 286–304.
[46] J. Chapman, R. Kireev, C. Nester, and P. Wadler. 2019. System F in Agda, for fun and profit. In *Proceedings of the MPC*. Springer International Publishing, 255–297.
[47] K. Chatterjee, A. K. Goharshady, and Y. Velner. 2018. Quantitative analysis of smart contracts. In *Proceedings of the ESOP*, Vol. 10801 LNCS. Springer Verlag, 739–767. arXiv:1801.03367.
[48] H. Chen, M. Pendleton, L. Njilla, and S. Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.* 53, 3 (June 2020).
[49] J. Chen, X. Xia, D. Lo, and J. Grundy. 2020. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. arXiv:2005.07908.
[50] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang. 2020. SODA: A generic online detection framework for smart contracts. In *Proceedings of the NDSS*.
[51] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *Proceedings of the IEEE SANER*. 442–446.
[52] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. 2019. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the ACM CCS*. ACM, 1503–1520.
[53] W. Chen, T. Zhang, Z. Chen, Z. Zheng, and Y. Lu. 2020. Traveling the token world: A graph analysis of ethereum ERC20 token ecosystem. In *Proceedings of the WWW*. ACM, 1411–1421.

[54] C. D. Clack and G. Vanca. 2018. Temporal aspects of smart contracts for financial derivatives. In *Proceedings of the ISoLA*, Vol. 11247 LNCS. Springer Verlag, 339–355. arXiv:1805.11677.
[55] M. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B. A. Myers, J. Sunshine, and J. Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. arXiv:1909.03523.
[56] C. Colombo, J. Ellul, and G. J. Pace. 2018. Contracts over smart contracts: Recovering from violations dynamically. In *Proceedings of the ISoLA*, Vol. 11247 LNCS. Springer Verlag, 300–315.
[57] S. E. S. Crawford and E. Ostrom. 1995. A grammar of institutions. *Am. Polit. Sci. Rev.* 89, 3 (1995), 582–600.
[58] L. P. A. da Horta, J. S. Reis, M. Pereira, and S. M. de Sousa. 2020. WhylSon: Proving your Michelson Smart Contracts in Why3. arXiv:2005.14650.
[59] M. di Angelo and G. Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *Proceedings of the IEEE DAPPCON*. IEEE, 69–78.
[60] T. Dickerson, P. Gazzillo, M. Herlihy, V. Saraph, and E. Koskinen. 2019. Proof-carrying smart contracts. In *Proceedings of the FC*. Springer Berlin, 325–338.
[61] W. Duo, H. Xin, and M. Xiaofeng. 2020. Formal analysis of smart contract based on colored petri nets. *IEEE Intell. Syst.* 35, 3 (2020), 19–30.
[62] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the IEEE/ACM ICSE*. arXiv:1910.10601.
[63] J. Ellul and G. J. Pace. 2018. Runtime verification of ethereum smart contracts. In *Proceedings of the EDCC*. IEEE, 158–163.
[64] J. Feist, G. Greico, and A. Groce. 2019. Slither: A static analysis framework for smart contracts. In *Proceedings of the IEEE/ACM WETSEB*. IEEE Press, 8–15.
[65] Y. Feng, E. Torlak, and R. Bodík. 2019. Precise attack synthesis for smart contracts. arXiv:1902.06067.
[66] C. Ferreira Torres, M. Baden, R. Norvill, and H. Jonker. 2019. ÆGIS: Smart shielding of smart contracts. In *Proceedings of the ACM CCS*. ACM, 2589–2591.
[67] C. Ferreira Torres, J. Schütte, and R. State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the ACSAC*. ACM, 664–676.
[68] F. Fournier and I. Skarbovsky. 2019. Enriching smart contracts with temporal aspects. In *Proceedings of the ICBC*, Vol. 11521 LNCS. Springer Verlag, 126–141.
[69] J. Frank, C. Aschermann, and T. Holz. 2020. ETHBMC: A bounded model checker for smart contracts. In *Proceedings of the USENIX Security*. USENIX Association.
[70] C. K. Frantz and M. Nowostawski. 2016. From institutions to code: Towards automated generation of smart contracts. In *Proceedings of the IEEE FAS-W*. IEEE, 210–215.
[71] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen. 2019. EASYFLOW: Keep ethereum away from overflow. In *Proceedings of the the IEEE/ACM ICSE-Companion*. 23–26.
[72] T. Genet, T. Jensen, and J. Sauvage. 2020. *Termination of Ethereum's Smart Contracts*. Research Report. Univ Rennes, Inria, CNRS, IRISA. Retrieved from https://hal.inria.fr/hal-02555738.
[73] L. M. Goodman. 2014. Tezos — A Self-Amending Crypto-Ledger. White Paper. Retrieved from https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.
[74] G. Governatori. 2014. Thou shalt is not you will. *CoRR* abs/1404.1685 (2014).
[75] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artif. Intell. Law* 26, 4 (01 Dec. 2018), 377–409.
[76] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. 2019. Gigahorse: Thorough, declarative decompilation of smart contracts. In *Proceedings of the IEEE/ACM ICSE*. 1176–1186.
[77] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. 2018. MadMax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proceedings of the ACM OOPSLA*.
[78] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the POST*. Springer, Cham, 243–269.
[79] A. Groce, J. Feist, G. Grieco, and M. Colburn. 2019. What are the Actual Flaws in Important Smart Contracts (and How Can We Find Them)? arXiv:1911.07567.
[80] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM POPL*.
[81] NCC Group. 2018. TOP 10 — Arithmetic Issues. Retrieved from https://www.dasp.co/.
[82] S. Haber and W. S. Stornetta. 1991. How to time-stamp a digital document. *J. Cryptol.* 3, 2 (Jan. 1991), 99–111.
[83] Á. Hajdu and D. Jovanović. 2019. solc-verify: A modular verifier for solidity smart contracts. arXiv:1907.04262.
[84] Á. Hajdu, D. Jovanović, and G. Ciocarlie. 2020. Formal Specification and Verification of Solidity Contracts with Events. arXiv:2005.10382.

[85] D. Harz and W. Knottenbelt. 2018. Towards safer smart contracts: A survey of languages and verification methods. arXiv:1809.09805.
[86] J. He, M. Balunovic̀, N. Ambroladze, P. Tsankov, and M. Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the ACM CCS*. ACM, 531–548.
[87] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, and X. Jiang. 2020. Security Analysis of EOSIO Smart Contracts. arXiv:2003.06568.
[88] Y. Hirai. 2016. Formal Verification of Deed Contract in Ethereum Name Service. Retrieved from https://yoichihirai.com/deed.pdf.
[89] Y. Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *Proceedings of the FC*. Springer, Cham, 520–535.
[90] Y. Hirai. 2018. Blockchains as Kripke models: An analysis of atomic cross-chain swap. In *Proceedings of the ISoLA*. Springer International Publishing, 389–404.
[91] G. J. Holzmann. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (1997), 279–295. DOI: https://doi.org/10.1109/32.588521
[92] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi. 2019. Smart contract security: A software lifecycle perspective. *IEEE Access* 7 (2019), 150184–150202.
[93] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng. 2019. Recommending differentiated code to support smart contract update. In *Proceedings of the IEEE/ACM ICPC*. 260–270.
[94] A. Imeri, N. Agoulmine, and D. Khadraoui. 2020. Smart contract modeling and verification techniques: A survey. In *Proceedings of the ADVANCE*. 1–8.
[95] Runtime Verification Inc. 2018. ERC20-K: Formal Executable Specification of ERC20. Retrieved from https://github.com/runtimeverification/erc20-semantics.
[96] J. Jiao, S. Kan, S.-W. Lin, D. Sanán, Y. Liu, and J. Sun. 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *Proceedings of the IEEE S&P*. IEEE Computer Society, 1265–1282.
[97] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. 2018. ZEUS: Analyzing safety of smart contracts. In *Proceedings of the NDSS*.
[98] T. Kasampalis, D. Guth, B. Moore, T. F. Serbanuta, Y. Zhang, D. Filaretti, V. Serbanuta, R. Johnson, and G. Roşu. 2019. IELE: A rigorously designed language and tool ecosystem for the blockchain. In *Proceedings of the FM*.
[99] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the ACM ISSTA*. ACM, 363–373.
[100] J. Kongmanee, P. Kijsanayothin, and R. Hewett. 2019. Securing smart contracts in blockchain. In *Proceedings of the ACM/IEEE ASEW*. 69–76.
[101] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE S&P*. 839–858.
[102] J. Krupp and C. Rossow. 2018. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the USENIX Security*. ACM, 1317–1333.
[103] A. Juels L. Breidenbach, P. Daian and E. G. Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. Retrieved from https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.
[104] J. Ladleif and M. Weske. 2019. A unifying model of legal smart contracts. In *Proceedings of the ER*. 323–337.
[105] P. Lamela Seijas, A. Nemish, D. Smith, and S. Thompson. 2020. Marlowe: Implementing and analysing financial contracts on blockchain. Retrieved from https://iohk.io/en/research/library/papers/marloweimplementing-and-analysing-financial-contracts-on-blockchain/.
[106] C. Laneve, C. S. Coen, and A. Veschetti. 2019. *On the Prediction of Smart Contracts' Behaviours*. Springer International Publishing, 397–415.
[107] A. Li, J. A. Choi, and F. Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the ACM PLDI*. ACM, 438–453.
[108] A. Li and F. Long. 2018. Detecting standard violation errors in smart contracts. arXiv:1812.07702.
[109] X. Li, Z. Shi, Q. Zhang, G. Wang, Y. Guan, and N. Han. 2019. Towards verifying ethereum smart contracts at intermediate language level. In *Proceedings of the ICFEM*. Springer International Publishing, 121–137.
[110] X. Li, C. Su, Y. Xiong, W. Huang, and W. Wang. 2019. Formal verification of BNB smart contract. In *Proceedings of the the BIGCOM*. 74–78.
[111] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. 2018. ReGuard: Finding reentrancy bugs in smart contracts. In *Proceedings of the IEEE/ACM ICSE*. ACM, 65–68.
[112] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun. 2018. S-gram: Towards semantic-aware security auditing for ethereum smart contracts. In *Proceedings of the ACM/IEEE ASE*. ACM, 814–819.
[113] Y. Liu, Y. Li, S.-W. Lin, and R. Zhao. 2020. Towards automated verification of smart contract fairness. In *Proceedings of the ACM ESEC/FSE*.

[114] Y. Liu, J. Sun, and J. S. Dong. 2011. PAT 3: An extensible architecture for building multi-domain model checkers. In *Proceedings of the IEEE ISSRE*.
[115] Z. Liu and J. Liu. 2019. Formal verification of blockchain smart contract based on colored petri net models. In *Proceedings of the IEEE COMPSAC*. IEEE, 555–560.
[116] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making smart contracts smarter. In *Proceedings of the ACM CCS*. ACM, 254–269.
[117] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi. 2019. EVM*: From offline detection to online reinforcement for ethereum virtual machine. In *Proceedings of the IEEE SANER*. 554–558.
[118] G. Madl, L. Bathen, G. Flores, and D. Jadav. 2019. Formal verification of smart contracts using interface automata. In *Proceedings of the IEEE Blockchain*. 556–563.
[119] D. Magazzeni, P. Mcburney, and W. Nash. 2017. Validation and verification of smart contracts: A research agenda. *Computer* 50, 9 (2017), 50–57.
[120] D. B. Maksimov, I. A. Yakimov, and A. S. Kuznetsov. 2020. Statistical model checking for blockchain-based applications. *IOP Conf. Ser. Mater. Sci. Eng.* 734 (Jan. 2020), 012152.
[121] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey. 2019. VeriSolid: Correct-by-design smart contracts for ethereum. arXiv:1901.01292.
[122] A. Miller, Z. Cai, and S. Jha. 2018. Smart contracts and opportunities for formal methods. In *Proceedings of the ISoLA*, Vol. 11247 LNCS. Springer Verlag, 280–299.
[123] C. Molina-Jimenez, I. Sfyrakis, E. Solaiman, I. Ng, M. Weng Wong, A. Chun, and J. Crowcroft. 2018. Implementation of smart contracts using hybrid architectures with on and off-blockchain components. In *Proceedings of the IEEE SC2*. 83–90.
[124] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the ACM/IEEE ASE*. 1186–1189.
[125] B. Mueller. 2018. *Smashing Ethereum Smart Contracts for Fun and Real Profit*. Technical Report.
[126] S. Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved July 14, (2020) from https://bitcoin.org/bitcoin.pdf/. (2008).
[127] Z. Nehai and F. Bobot. 2019. Deductive proof of ethereum smart contracts using why3. arXiv:1904.11281.
[128] Z. Nehai, P. Piriou, and F. Daumas. 2018. Model-checking of smart contracts. In *Proceedings of the IEEE iThings/GreenCom/CPSCom/SmartData*. IEEE, 980–987.
[129] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka. 2020. Verified development and deployment of multiple interacting smart contracts with verisolid. Retrieved from http://www.eecg.utoronto.ca/~veneris/20ICBC.pdf.
[130] J. B. Nielsen and B. Spitters. 2019. Smart Contract Interactions in Coq. arXiv:1911.04732.
[131] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. *Proceedings of the ACSAC*.
[132] R. O'Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the ACM PLAS*. ACM, 107–120.
[133] OpenZeppelin. 2020. SafeMath Library. Retrieved from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.
[134] R. M. Parizi, A. Singh, and A. Dehghantanha. 2018. Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *Proceedings of the ICBC*, Vol. 10974 LNCS. Springer Verlag, 75–91.
[135] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. 2018. A formal verification tool for ethereum VM bytecode. In *Proceedings of the ACM ESEC/FSE*. ACM, 912–915.
[136] D. Perez and B. Livshits. 2019. Smart contract vulnerabilities: Does anyone care? arXiv:1902.06710.
[137] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. 2020. VerX: Safety verification of smart contracts. In *Proceedings of the IEEE S&P*. IEEE Computer Society, 414–430.
[138] S. Popejoy. 2017. The Pact Smart-Contract Language. Retrieved from https://www.kadena.io/whitepapers.
[139] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. 2019. Security analysis methods on ethereum smart contract vulnerabilities: A survey. arXiv:1908.08605.
[140] D. Prechtel, T. Groß, and T. Müller. 2019. Evaluating spread of "gasless send" in ethereum smart contracts. In *Proceedings of the IFIP NTMS*. 1–6.
[141] M. Qu, X. Huang, X. Chen, Y. Wang, X. Ma, and D. Liu. 2018. Formal verification of smart contracts from the perspective of concurrency. In *Proceedings of the SmartBlock*, Vol. 11373 LNCS. Springer Verlag, 32–43.
[142] RChain. 2018. Contract Design — RChain Architecture 0.9.0 documentation. Retrieved from https://architecture-docs.readthedocs.io/contracts/contract-design.html.
[143] J. S. Reis, P. Crocker, and S. M. de Sousa. 2020. Tezla, an Intermediate Representation for Static Analysis of Michelson Smart Contracts. arXiv:2005.11839.
[144] J. Rushby. 2001. *Theorem Proving for Verification*. Springer Berlin, 39–57.

[145] N. F. Samreen and M. H. Alalfi. 2020. Reentrancy vulnerability identification in ethereum smart contracts. In *Proceedings of the IEEE IWBOSE*. 22–29.
[146] D. C. Sánchez. 2018. Raziel: Private and verifiable smart contracts on blockchains. *CoRR* abs/1807.09484 (2018).
[147] N. Sato, T. Tateishi, and S. Amano. 2018. Formal requirement enforcement on smart contracts based on linear dynamic logic. In *Proceedings of the IEEE iThings/GreenCom/CPSCom/SmartData*. 945–954.
[148] S. Sayeed, H. Marco-Gisbert, and T. Caira. 2020. Smart contract: Attacks and protections. *IEEE Access* 8 (2020), 24416–24427.
[149] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. arXiv:2005.06227.
[150] F. Schrans, S. Eisenbach, and S. Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Proceedings of the ACM Programming Companion*. ACM, 218–219.
[151] I. Sergey and A. Hobor. 2017. A concurrent perspective on smart contracts. In *Proceedings of the FC*, Vol. 10323 LNCS. Springer Verlag, 478–493.
[152] I. Sergey, A. Kumar, and A. Hobor. 2018. Temporal properties of smart contracts. In *Proceedings of the ISoLA*. Springer, Cham, 323–338.
[153] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. 2019. Safer smart contract programming with Scilla. *Proceedings of the ACM OOPSLA*.
[154] E. Shishkin. 2018. Debugging smart contract's business logic using symbolic model-checking. arXiv:1812.00619.
[155] D. Siegel. 2016. Understanding The DAO Attack — CoinDesk. Retrieved from https://www.coindesk.com/understanding-dao-hack-journalists.
[156] A. Singh, R. Parizi, Q. Zhang, K.-K. R. Choo, and A. Dehghantanha. 2019. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Comput. Secur.* 88 (10 2019), 101654.
[157] S. So, M. Lee, J. Park, H. Lee, and H. Oh. 2020. VeriSmart: A highly precise safety verifier for ethereum smart contracts. In *Proceedings of the IEEE S&P*. IEEE Computer Society, 825–841.
[158] F. Spoto. 2020. Enforcing determinism of Java smart contracts. In *Proceedings of the FC*, Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala (Eds.). Springer International Publishing, Cham, 568–583.
[159] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev. 2019. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the ACM CCS*. ACM, 1759–1776.
[160] J. Sun, Y. Liu, and J. S. Dong. 2008. Model checking CSP revisited: Introducing a process analysis toolkit. In *Proceedings of the ISoLA*. Springer Berlin.
[161] T. Sun and W. Yu. 2020. A formal verification framework for security issues of blockchain smart contracts. *Electronics* 9, 2 (Feb. 2020), 255.
[162] D. Suvorov and V. Ulyantsev. 2019. Smart contract design meets state machine synthesis: Case studies. arXiv:1906.02906.
[163] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Mond.* 2, 9 (Sep. 1997).
[164] Parity Technologies. 2017. A Postmortem on the Parity Multi-Sig Library Self-Destruct — Parity Technologies. Retrieved from https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
[165] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. 2018. SmartCheck: Static analysis of ethereum smart contracts. In *Proceedings of the IEEE/ACM WETSEB*. 9–16.
[166] P. Tolmach, Y. Li, S.-W. Lin, and Y. Liu. 2021. Formal Analysis of Composable DeFi Protocols. arXiv:2103.00540.
[167] C. Ferreira Torres, M. Steichen, and R. State. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *Proceedings of the USENIX Security*. USENIX Association, 1591–1607.
[168] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the ACM CCS*. ACM, 67–82.
[169] R. v. d. Meyden. 2019. On the specification and verification of atomic swap smart contracts (extended abstract). In *Proceedings of the IEEE ICBC*. 176–179.
[170] E. Viglianisi, M. Ceccato, and P. Tonella. 2020. A federated society of bots for smart contract testing. *J. Syst. Softw.* 168 (2020), 110647.
[171] F. Vogelsteller and V. Buterin. 2015. ERC-20 Token Standard. Retrieved from https://eips.ethereum.org/EIPS/eip-20.
[172] H. Wang, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu. 2019. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. arXiv:1909.06605.
[173] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu. 2019. VULTRON: Catching vulnerable smart contracts once and for all. In *Proceedings of the IEEE/ACM ICSE*. IEEE Press, 1–4.
[174] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang. 2019. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Trans. Syst., Man, Cybern. Syst.* (Feb. 2019), 1–12.

[175] S. Wang, C. Zhang, and Z. Su. 2019. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM OOPSLA*.
[176] Y. Wang, S. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. Retrieved from https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain/.
[177] K. Weiss and J. Schütte. 2019. Annotary: A concolic execution system for developing secure smart contracts. In *Proceedings of the ESORICS*. Springer International Publishing, 747–766.
[178] M. Wohrer and U. Zdun. 2018. Design patterns for smart contracts in the ethereum ecosystem. In *Proceedings of the IEEE iThings/GreenCom/CPSCom/SmartData*. IEEE, 1513–1520.
[179] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ether. Proj. Yell. Paper* 151 (2014), 1–32.
[180] W. Xu and G. A. Fink. 2019. Building Executable Secure Design Models for Smart Contracts with Formal Methods. arXiv:1912.04051.
[181] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber. 2018. A pattern collection for blockchain-based applications. In *Proceedings of the ACM EuroPLoP*. ACM, ACM.
[182] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. 2019. Potential risks of hyperledger fabric smart contracts. In *Proceedings of the IEEE IWBOSE*. 1–10.
[183] Z. Yang and H. Lei. 2018. Lolisa: Formal syntax and semantics for a subset of the solidity programming language. *CoRR* abs/1803.09885 (2018).
[184] Z. Yang, H. Lei, and W. Qian. 2019. A hybrid formal verification system in Coq for ensuring the reliability and security of ethereum-based service smart contracts. *CoRR* abs/1902.08726 (2019).
[185] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury. 2019. Smart Contract Repair. arXiv:1912.05823.
[186] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. 2016. Town Crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM CCS*. ACM, 270–282.
[187] X. Zhang, Y. Li, and M. Sun. 2020. Towards a formally verified EVM in production environment. In *Proceedings of the COORDINATION*. Springer International Publishing, 341–349.
[188] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Fut. Gen. Comput. Syst.* 105 (2020), 475–491.
[189] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara. 2018. Security assurance for smart contract. In *Proceedings of the IFIP NTMS*. 1–5.
[190] J. Zhu, K. Hu, M. Filali, J.-P. Bodeveix, and J.-P. Talpin. 2020. Formal Verification of Solidity contracts in Event-B. arXiv:2005.01261.
[191] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Trans. Softw. Eng.* (Sep. 2019), 1–1.
[192] N. Zupan, P. Kasinathan, J. Cuellar, and M. Sauer. 2020. *Secure Smart Contract Generation Based on Petri Nets*. Springer Singapore, 73–98.

Received July 2020; revised April 2021; accepted April 2021