High Level Synthesis and Verification of Solidity Smart Contracts

Yuntao Cai*, Keerthi Nelaturu*, Andreas Veneris*,†
* Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
†Department of Computer Science, and Munk School of Global Affairs & Public Policy, University of Toronto
yuntao.cai@mail.utoronto.ca, nelaturuk@gmail.com, veneris@eecg.toronto.edu

***Abstract*—The Ethereum Virtual Machine (EVM) allows decentralized applications to operate without a trusted intermediary through smart contracts: self-executing programs that encode arbitrarily complex on-chain logic. Yet non-programmer domain experts who wish to build such decentralized applications still face two obstacles: smart-contract development demands specialized programming knowledge they typically lack, and deployed contracts are often harbor surprising vulnerabilities. These barriers call for a development model that separates domain expertise from low-level contract engineering. Building on that separation, this paper presents a framework where programmers define schemas and reusable constructs for a given domain, which domain experts can then use to specify smart contracts in familiar terms and concepts rather than in a specialized programming language. Expert-authored models are checked against behavioral properties before automatic synthesis into Solidity. We present a prototype implementation of this approach and validate it on two domains across multiple contracts. Our evaluation demonstrates considerable similarity between the generated contracts and existing contracts sourced from GitHub and Etherscan, with an average gas consumption reduction of 53 percent, and confirms that the specified safety and liveness properties hold on each domain expert created contract.**

***Keywords*—Smart contracts, DSL, Synthesis, Verification**

I. Introduction

The concept of a peer-to-peer electronic currency was first formalized by Nakamoto [1], whose Bitcoin protocol demonstrated that trustless financial transactions could be conducted without a central authority. Building on this foundation, Buterin [2] introduced Ethereum, which extended the blockchain paradigm beyond simple currency transfers with the Ethereum Virtual Machine (EVM), where developers deploy *smart contracts*—self-executing programs that enable transparent, trust-minimized execution of arbitrarily complex on-chain logic. This capability has given rise to an entire ecosystem of decentralized applications, including supply-chain tracking [3], decentralized finance [4], and asset tokenization [5], among others. Despite these advancements, smart contract development remains difficult in practice because these contracts are typically written in specialized programming languages such as Solidity [6]. Non-programmers seeking to develop blockchain applications can often articulate contractual intent, yet struggle to produce correct, vulnerability-free implementations. Widely publicized incidents such as the DAO and Parity compromises [7,8] show that small errors in code can cause large financial losses, and because deployed contracts are immutable, remediation is often costly or impractical. While formal methods and automated verifiers for Solidity have made notable progress [9]–[14], much of current practice remains post-hoc, is tool-specific, and difficult for non-specialists to apply consistently at scale.

A practical response to these difficulties is to raise the level of abstraction using small, purpose-built languages tailored to particular domains, known as domain-specific languages (DSLs). Prior DSL-oriented work [15]–[19] follows this strategy, but each approach is tied to its own notation and target domain. The resulting languages are often still developer-oriented, so non-programmers may struggle to use them, and users who span multiple application areas must adopt disjoint toolchains and non-transferable skills.

We propose a high-level synthesis framework in which a *programmer*—a person proficient in software development—defines a domain-specific language for a chosen field, such as gaming, legal workflows, or NFT markets. A *domain expert*—a person with substantial knowledge of that field, but without proficiency in smart-contract programming or formal verification—then specifies smart contracts in this notation rather than in a specialized programming language. Behavioral properties can be stated by domain experts in near-natural language and verified against the high-level model, so experts can refine the specification and re-check safety and liveness properties before any implementation is produced. Verified models are then automatically compiled into Solidity. The same framework can thus serve multiple application areas while remaining approachable for non-programmers.

Across ten case studies, generated contracts show moderate similarity between generated and reference contracts, and significant reduction in deployment gas and bytecode size.

In summary, this paper makes the following contributions:

- We propose framework where programmers can define DSLs in their chosen field of interest.
- We enables domain experts to efficiently create smart contracts within their area of expertise without being exposed to low level code.
- We evaluate our approach through syntactic comparison and property verification, showing structural alignment with references, favorable deployment cost, and successful automated verification of domain expert–authored safety and liveness properties.

II. BACKGROUND AND RELATED WORK

A. Domain-Specific Languages

DSL is defined as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [20]. In prior work, Frantz and Nowostawski [15] focus on institutional contracts, expressing rules in ADICO form (Attributes, Deontic, Aim, Conditions, Or else). Wöhrer and Zdun [16] focus on legal contracts and propose Contract Modeling Language, an object-oriented DSL that models contractual building blocks such as parties, assets, and covenants through clauses and actions over shared state. Oei et al. [17] propose Psamathe for asset transfers centered on *flows*—atomic moves of typed assets between variables—placed inside *transformers* (operations that update contract state) under rules meant to prevent common asset-handling errors. A key limitation of these approaches, and of related single-purpose DSLs such as Nomos [18] for asset tracking and Peyton Jones–style financial contracts [19], is that each is a fixed DSL solution specialized to one modeling concern. As shown in Figure 1, the three notations are visibly disjoint in vocabulary, control structure, and intended semantics, so an institutional ADICO model cannot be reused inside CML or Psamathe without custom translation layers or re-engineered tooling. The result is notation fragmentation and high integration overhead in cross-domain smart contract development. The same fragments also show that each DSL still demands a specialized syntax and conceptual model, which can remain daunting for non-programmers. The same fragments also show that each DSL still requires non-programmers to learn a specialized syntax and conceptual model, which can remain daunting and forces a new notation to be learned for each domain.

B. Verification

It is essential to establish a contract's security before deployment, especially when the contract will be entrusted with cryptocurrencies. Formal verification has made important progress, but each approach has practical limitations. Correct-by-design frameworks such as VeriSolid [9] represent contracts as graphical transition systems and check safety, liveness, and deadlock-freedom properties with temporal-logic templates. We adopt VeriSolid as our verification backend because it provides this model-level checking pipeline. Nevertheless, VeriSolid remains oriented toward developers who must work in both its transition-system formalism and Solidity within the model, so non-experts cannot author contracts or properties purely from domain concepts. Celestial [10] lets programmers annotate Solidity with Hoare-style specifications, translates contracts and specs into $F^\star$, and verifies them against an $F^\star$ model of blockchain semantics before erasing the specs for deployment. The workflow therefore presumes Solidity authorship plus the ability to write and discharge $F^\star$-oriented specifications. ESBMC-Solidity [11] analyzes existing Solidity via the compiler AST, lowers it to an intermediate representation, and uses bounded/symbolic model checking with SMT solvers, typically with assertions and assumptions embedded in the source. Related post-hoc workflows [12] likewise start from Solidity and require formal annotations that non-specialists struggle to produce. Moreover, none of these approaches capture high-level domain intent in a model that non-programmer experts can author directly. Other work includes ERC/NFT-focused verification [13] and code-pruning support for Solidity static analysis [14], both narrow in scope and tied to a specific domain or specialized tool, with limited cross-domain interoperability.

(a)

```
Adico(
  A("system"),
  D(must),
  I("release", object("objectOfInterest"),
    target("buyer", "address")),
  C(IF("msg.value", Operator.>=, "price")))
```

(b)

```
action Boolean becomeRichest(TokenTransaction t)
   caller.deposit(t.amount)
   if(t.amount > mostSent)
      transfer(richest, token.quantity)
      richest = caller
      mostSent = t.amount
      return true
   return false
```

(c)

```
transformer giveRightToVote(this : Election, voter : address) {
  only when msg.sender = this.chairperson
  new Voter(voter) --> this.eligibleVoters
}
```

Fig. 1. Illustrative snippets from three smart-contract DSLs: (a) ADICO-style institutional rules [15]; (b) CML for legal contracts with clauses and actions [16]; and (c) Psamathe's flow-oriented asset DSL [17].

III. Proposed Methodology and Design

Our approach consist of four components which provides a comprehensive pipeline from high-level specification to executable code:

1) **DSL Definition Layer**—where the programmer defines domain vocabulary, operations, constructs experts can use, and syntax of the DSL;
2) **Contract Specification Layer**—where the domain expert models contract behavior as states, conditional transitions, and expert-defined functions built from the programmer-supplied vocabulary;
3) **Verification Layer**—where safety and liveliness properties over transitions and statements are checked on the model;
4) **Solidity Code Generation Layer**—where the verified model is lowered to deployable Solidity with traceable states and transitions.

We describe our implementation of these layers in Section E.

A. DSL Definition

At the conceptual level, a Solidity contract consists of persistent on-chain variables together with a set of functions that read or update those variables. The programmer determines which persistent variables a domain requires and which operations experts may use to read or update them. To that end, our framework provides three primitive types—`integer`, `user`, and `bytes`—that the programmer can use to declare variables. These primitives may be renamed to domain-friendly labels (e.g., an `integer` as `price` or `wager`) or assembled into higher-level types, such as structs like `Card` or mappings such as a `CollectionOfBids`. For functions, the platform supplies general-purpose operations commonly used across decentralized applications (e.g., Ether transfers, encoding and decoding of data, randomness). The programmer then defines additional domain operations that experts may invoke when modeling contracts. For example, a game DSL may introduce operations such as `dealCard` and `dealCardFaceDown`, whereas an NFT marketplace DSL may expose operations such as `transferToken` and `computeProfitWithFee`.

B. Contract Specification

This layer enables domain experts to define contract behavior from the programmer-supplied building blocks. Rather than writing Solidity directly, experts assemble those constructs into an executable model of the contract. We represent that model similarly to a finite state machine: states correspond to phases of the contract life cycle, and transitions describe how the system progresses between those phases. State 0 is the initial state (constructor). When the modeled workflow is meant to complete, experts may add a terminating state in which the contract stops accepting further transitions. Domain experts then declare the internal states (e.g., *dealCardsToPlayer*, *playerHitOrStand*, *dealerPlay*) and directed transitions between them.

Each transition specifies a collection of conditions that must be satisfied to move from the current state to the state it points to. Some conditions are simple (e.g., that a buyer has agreed to a trade), while others require non-trivial evaluation (e.g., that a player's score does not exceed 21). We provide a fixed set of constructs with which domain experts may determine whether those conditions hold: local variables for intermediate results, conditional branches, arithmetic over domain variables, and calls to built-in or programmer-defined functions. Experts may also define functions of their own when the function is contract-specific. For instance, `calculateScore` differs across card games even if shared operations such as `dealCard` can be supplied by the programmer. Once the conditions hold, the model advances to the next state and applies any persistent-variable updates associated with the transition. Figure 2 shows an example state machine for a blackjack card game.

C. Verification of properties

We verify the contract model before any Solidity is generated: deployment code is emitted only after the model is accepted. Our verification layer follows VeriSolid [9], in which contracts are modeled as transition systems and safety and liveness properties are stated via templates over transitions and transition statements. We reuse that template-based checking style, however the verification is not applied to a single developer-built model. Instead, it is the expert-authored lifecycle model defined above, constructed from a programmer-defined domain package. Properties are to be expressed in the following templates:

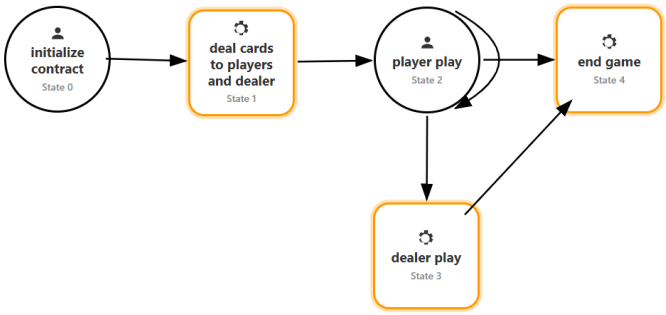

Fig. 2. Domain Expert Defining States

$\langle Transitions \cup Statements \rangle$ cannot happen after $\langle Transitions \cup Statements \rangle$.

If $\langle Transitions \cup Statements \rangle$ happens, $\langle Transitions \cup Statements \rangle$ can happen only after $\langle Transitions \cup Statements \rangle$ happens.

$\langle Transitions \cup Statements \rangle$ will eventually happen after $\langle Transitions \cup Statements \rangle$.

The first template forbids an event once another has occurred. The second template constrains the order in which events may occur after a triggering event. The third template requires that an event must occur eventually after another (liveness).

In VeriSolid, a *transition* is a named edge of the developer's transition system and is implemented as a Solidity function; a *statement* is an individual instruction inside a transition's action, written `Transition.Statement`. Our expert models use the dual presentation: each lifecycle *state* (node) is compiled to one Solidity function whose guards and body come from that node and its outgoing edges. Consequently, for a state with a single successor, the VeriSolid transition atom is that generated function (e.g., HighLow's `playerPlay`). When a state has several outgoing edges—as in Blackjack's `playerPlay`, which may move to `dealer_play`, remain in `player_play`, or enter `End_game`—one function denotes several transitions. We therefore admit the branch atom $t{\to}s$, the execution of t that reaches state s. Because each branch ends in a distinct `currentState` assignment, $t{\to}s$ corresponds to the VeriSolid statement atom t`.currentState = State.`s`;`,

which the augmented transition system already exposes. Finer statement atoms may likewise name other emitted steps from an edge's computation (e.g., a helper call inside the function body).

D. Solidity Code Generation

Once the domain expert has specified the contract model and its behavioral properties, and verification has succeeded, the model is translated into Solidity. Every model variable is drawn from the programmer-defined DSL (itself based on the framework's primitive types) and therefore has a direct Solidity counterpart—scalar fields, structs, and registry/mapping layouts included. Programmer-defined domain operations and expert-defined helper routines are likewise emitted as callable functions in the generated contract, together with any shared runtime helpers they invoke (for example, escrow wallet primitives). The expert's finite-state machine is lowered as a control-flow skeleton. Contract phases are represented by a Solidity `enum` and a `currentState` variable. Each lifecycle state becomes a function that (i) requires the machine to be in that state, (ii) enforces the state's guards and parameter checks, (iii) executes the state's computation steps, and (iv) selects among outgoing transitions in declaration order. If a transition's guard evaluates to true, its action is executed and `currentState` is updated to the destination; otherwise the next outgoing transition is tried. If none qualifies, execution reverts. States marked for automatic continuation are emitted as `internal` functions and may be invoked immediately at the end of a successful transition, so a single external call can advance several phases in one transaction. Terminal states are absorbing: once the machine enters an end state, it does not return to an earlier phase. The result is a single, self-contained contract in which states, transitions, guards, helpers, and storage can be traced from the expert model to the generated Solidity identifiers used later when binding properties for checking.

E. Implementation

We implement the proposed methodology as a web-based prototype. The programmer interface defines a DSL; the expert interface allows domain experts to model contract logic and specify behavioral properties. Only after the model passes verification does the generator combine these artifacts into a single self-contained `.sol` file. Failed checks return a counterexample trace to the expert for revision.

Figure 3 gives a visual representation of the pipeline. Arrows labeled $\leftarrow$ in the figure represents the assignment operator. Σ is the programmer-defined *type vocabulary* (primary and composite variables). $\mathcal{F}$ is the catalog of *domain operations* experts may use in their contract. Together they form the published DSL $\mathcal{P} \leftarrow (\Sigma, \mathcal{F})$. The domain expert then creates a contract *state-machine model* $\mathcal{M}$ and a set of behavioral properties Φ. The verifier translates $\mathcal{M}$ into a formal transition system $\mathcal{T}$ and Φ into CTL formulas; if checking succeeds, codegen emits the Solidity contract C. We validated the prototype on ten contracts across two domains:

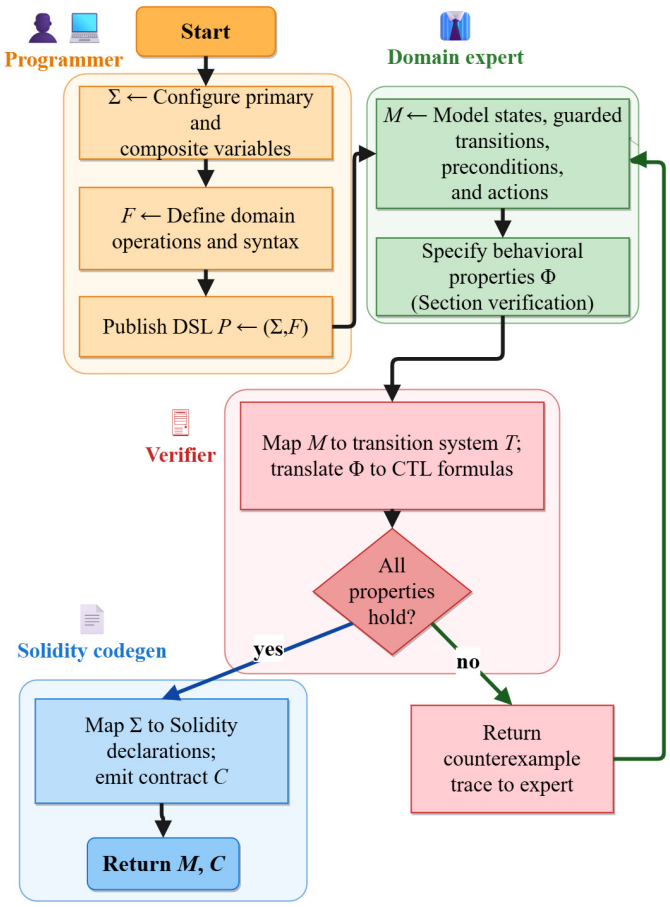

Fig. 3. Contract development and code generation pipeline

five card games and five NFT-marketplaces. All ten compiled without errors under Solidity ^0.8.20.

IV. Experimental Evaluations

We evaluate the prototype along two dimensions. First, we compare the syntactic structure, deployment cost, bytecode size, and manually authored behavioral tests of generated contracts against reference implementations sourced from online open-source repositories. Second, we validate domain-specific safety and liveness properties on the expert-authored models using the verification tool. Reference sources and the full property list are available in the project repository. [1]

A. Dataset Collection

We evaluate ten generated contracts against open-source references: five card games (High-Low, Blackjack, Baccarat, Dragon Tiger, Goldfish) and five NFT marketplace patterns (Direct sale, Auction, Blind Auction, Bartering, Mystery Box). References were drawn from GitHub and Etherscan; when several candidates existed, we chose the one whose gameplay or trading workflow best matched the generated design.

[1]Project repository: https://github.com/cai-yuntao/High-level-synthesis-and-verification-of-smart-contracts

Contract	API	AST	GasC	GasB	BCC	BCB	Paired
HighLow	10.0	93.5	699	2550	2925	5945	64.7
Blackjack	9.5	73.0	378	3252	1562	14645	52.9
Baccarat	9.7	82.0	1359	4184	5976	19131	35.3
Dragon Tiger	7.7	96.4	684	662	2857	2822	58.8
Goldfish	9.1	85.5	2624	1905	11824	3663	15.8
NFT Sale	42.3	98.0	812	835	3347	3591	85.7
NFT Auction	6.9	95.8	864	1753	3592	8107	63.2
NFT Blind	2.7	95.7	1078	2204	4581	9900	36.8
NFT Barter	2.9	92.4	671	2357	2796	11191	27.8
NFT Mystery	5.1	92.4	775	1522	3182	8640	22.2
Avg	**10.6**	**90.5**	**994**	**2122**	**4264**	**8764**	**46.3**

TABLE I
MULTI-CONTRACT COMPARISON (C = GENERATED; B = REFERENCE). **API** = PUBLIC SIGNATURE OVERLAP (%); **AST** = AST SHAPE SIMILARITY (%). **GASC/GASB** = DEPLOYMENT GAS FOR C/B (K GAS); **BCC/BCB** = DEPLOYED BYTECODE SIZE FOR C/B (KB). **PAIRED** = SHARE OF SCENARIOS WITH A PASSING BILATERAL TEST (%). GAS AND BYTECODE VALUES ARE ROUNDED TO KILOUNITS.

B. Comparison Metrics

Throughout, C denotes the generated contract and B the open-source reference. **API** measures overlap of the public callable surface: we normalize public function signatures and report the percentage shared by C and B. **AST** measures structural similarity at the syntax level: we compile each contract, record how often each node type appears in its abstract syntax tree, and compare the two frequency profiles with cosine similarity to see whether the contracts use similar mixes of Solidity constructs. **GasC** and **GasB** are deployment gas cost for C and B, respectively; **BCC** and **BCB** are the corresponding deployed bytecode sizes. We compile generated contracts with Solidity 0.8.20 and references with a compatible version (optimizer enabled, 200 runs), then deploy both on a local EVM.

The metrics above capture syntax and costs of deployment but do not establish semantic and behavioral equivalence. Reference contracts lack a machine-readable domain model, so we manually author a set of scenarios: we select prominent features, write executable unit tests for each contract, and count scenarios where both sides pass the test. **Paired** reports the percentage of scenarios in which both C and B are tested under the same scenario and that test passes. Although these metrics do not indicate formal equivalence between our generated contracts and the reference implementations, these unit tests nonetheless provide operational semantic agreement on a fixed behavioral checklist.

C. Results

Table I summarizes similarity, deployment cost, and scenario results for the ten generated–reference contract pairs. **API** overlap is low on average (10.6%) because each generated contract exposes only the compact interface implied by its editor state machine, while each reference exposes a fuller game- or marketplace-specific API (multiplayer coordination, commit–reveal flows, operator controls, native ETH payments, and similar extras). **AST** similarity is nevertheless high (91% on average), indicating that the generated contracts use a comparable mix of Solidity constructs even when surface entry points differ. Deployment gas averages 994k for C versus 2122k for B ($\approx 0.47\times$); bytecode averages 4264 B versus 8764 B ($\approx 0.49\times$). Generated contracts are consistently smaller because references implement those additional platform features—not because codegen is unusually lean on an identical feature set. The main exception is Goldfish, where our model is more elaborate than the chosen reference. Across the suite, pairs average about 16 scenario tests each, and **162/182** scenarios match their stated rules (≈89%). The **Paired** column (46.3% on average) is *not* a failure rate: it is the fraction of scenarios that are executable on *both* C and B under the same rule (cross comparisons). The remaining tests are intentionally candidate-only, baseline-only, or documented API differences, precisely where the reference exposes behav-

Contract	Property	Type
HighLow	If initialized, `playerGuessHighOrLow()` can only happen after `dealCardFaceDown(dealer)`.	Safety
Blackjack	`transferFund(player)` cannot happen after `calculateScoreBj (player.hand)>21`.	Safety
Baccarat	`dealCard(player)` will not happen after `calculateScoreBc (player. hand)==8 or calculateScoreBc (player.hand)==9`.	Safety
Dragon Tiger	If `dealDragon()`, `playerGuess()` can only happen after `dealTiger()`.	Safety
Goldfish	If `askRank(rankNumber)`, `transferCard()` can only happen after a `checkPlayerHave(rankNumber)`	Safety
NFT Sale	`transferItem()` cannot happen before `buyerConfirmPurchase()`.	Safety
NFT Auction	`placeBid()` cannot happen after `bid<highestBid`.	Safety
NFT Blind	`placeBlindBid()` cannot happen after `auctionEndTime<getMessageTime()`.	Safety
NFT Barter	`transferTokenBtoA()` cannot happen before `partyBAccept()`.	Safety
NFT Mystery	If `openBox()`, `reveal()` can only happen after `transferFund(player)`.	Safety
HighLow	`revealCard(hiddenCard)` will eventually happen after `playerGuessHighOrLow()`.	Liveness
Baccarat	`transferFund(player)` will eventually happen after `calculateScoreBc(player) >calculateScoreBc(dealer)`.	Liveness
Dragon Tiger	`revealCard(dragon)` will eventually happen after `dealDragon()`.	Liveness
NFT Auction	`refundLosingBidders()` will eventually happen after `auctionEndTime<getMessageTime()`.	Liveness
NFT Barter	`swapCompleted` will eventually happen `afterpartyBAccept() == 1` and `partyAAccept()==1`	Liveness

TABLE II
REPRESENTATIVE SAFETY AND LIVENESS PROPERTIES VERIFIED ON EXPERT MODELS.

ior with no counterpart in the generated FSM (for example, Hit/Stand Blackjack versus a single `playerPlay` transition, or commit–reveal HighLow versus an immediate guess). Where a shared story *can* be stated—phase guards, access control, stake limits, and core settlement—cross scenarios typically agree. Thus low API overlap and a mid-range Paired share are two views of the same design gap, while high AST similarity, roughly halved deployment cost, and high overall scenario match support that the generated contracts realize the expert lifecycle faithfully within that scoped interface.

D. Property Verification

Beyond executable scenarios, we check behavioral properties that domain experts can state without writing CTL by hand. Properties use VeriSolid's natural-language templates [9] over transitions and statements of the expert lifecycle model. In our setting, a transition is a generated state function (or a branch atom $t{\rightarrow}s$); a statement is an emitted step inside that function. A binding step maps each expert-facing atom to the corresponding Solidity identifier; unresolved atoms are rejected. The resulting CTL is discharged with VeriSolid/nuXmv on the generated contract. Table II lists representative properties for all ten case studies. Safety properties forbid illegal orderings (e.g., play before cards are dealt). Liveness properties require an internally triggered continuation after its predecessor, so the obligation does not depend on an external caller. These checks complement the scenarios: scenarios compare generated contracts to open-source references under concrete calls, while temporal properties constrain ordering and reachability in the expert lifecycle itself.

V. Conclusion

In this work, we propose a framework that separates domain knowledge from smart contract implementation. Programmers define reusable schemas, while domain experts author high-level models that are checked before automatic synthesis into smart contracts. We evaluate the prototype on ten generated-versus-reference contract pairs spanning card games and NFT marketplaces. Generated contracts are consistently smaller than their references, using roughly half the deployment gas and bytecode. Manually authored scenarios provide a semantic view beyond syntax: bilateral (**Paired**) coverage averages 46.3%. VeriSolid verification confirmed expected safety and liveness properties on the models at design time. Together, these results suggest that the toolchain can produce deployable contracts from verified expert models and support lightweight comparison against open-source baselines. Future work includes extending the approach to additional application domains and automating the generation of behavioral properties from the expert's model.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, accessed: Mar. 19, 2026. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," ethereum.org, 2014, accessed: May 2, 2026. [Online]. Available: https://ethereum.org/whitepaper/

[3] A. Sristy, "Blockchain in the food supply chain – what does the future look like?" 2021, accessed: Mar. 6, 2026. [Online]. Available: https://tech.walmart.com/content/walmart-global-tech/en_us/blog/post/blockchain-in-the-food-supply-chain.html

[4] Ethereum, "Decentralized finance (DeFi)," 2021. [Online]. Available: https://ethereum.org/en/defi/

[5] The Motley Fool, "This 1 accelerating trend could drive XRP and Ethereum higher and higher," 2026, accessed: Mar. 6, 2026. [Online]. Available: https://www.theglobeandmail.com/investing/markets/stocks/INTC/pressreleases/530414/this-1-accelerating-trend-could-drive-xrp-and-ethereum-higher-and-higher/

[6] Ethereum, "Ethereum," 2026. [Online]. Available: https://ethereum.org

[7] BBC News, "Hack attack drains start-up investment fund," 2016, accessed: Mar. 6, 2026. [Online]. Available: https://www.bbc.com/news/technology-36585930

[8] S. Palladino, "The parity wallet hack explained," OpenZeppelin Blog, 2017, accessed: May 3, 2026. [Online]. Available: https://www.openzeppelin.com/news/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[9] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Financial Cryptography and Data Security. FC 2019 International Workshops, Christ Church, Barbados, February 18–22, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 11598. Cham: Springer, 2019, pp. 446–465.

[10] S. Dharanikota, S. Mukherjee, C. Bhardwaj, A. Rastogi, and A. Lal, "Celestial: A smart contracts verification framework," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, New Haven, CT, USA, 2021, pp. 133–142.

[11] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, "ESBMC-Solidity: An SMT-based model checker for Solidity smart contracts," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Pittsburgh, PA, USA, 2022, pp. 65–69.

[12] X. Wang, X. Yang, and C. Li, "A formal verification method for smart contract," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, Xi'an, China, 2020, pp. 31–36.

[13] R. Ben Fekih, M. Lahami, M. Jmaiel, and S. Bradai, "Formal modeling and verification of ERC smart contracts: Application to NFT," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, Gammarth, Tunisia, 2023, pp. 556–561.

[14] Z. Zhou, Y. Xiong, W. Huang, and L. Ma, "SPrune: A code pruning tool for Ethereum Solidity contract static analysis," in *2020 6th International Conference on Big Data Computing and Communications (BIGCOM)*, Deqing, China, 2020, pp. 66–70.

[15] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Augsburg, Germany, 2016, pp. 210–215.

[16] M. Wöhrer and U. Zdun, "Domain specific language for smart contract development," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, Toronto, ON, Canada, 2020, pp. 1–9.

[17] R. Oei, "Psamathe: A DSL for safe blockchain assets," in *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Nov. 2020, pp. 16–18.

[18] A. Das, J. Hoffmann, and F. Pfenning, "Nomos: A protocol-enforcing, asset-tracking, and gas-aware language for smart contracts," in *Proc. 42nd Conf. Very Important Topics (CVIT)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, pp. 23:1–23:18, article 23.

[19] V. U. Wickramarachchi, C. I. Keppitiyagama, and K. G. Gunawardana, "Efficiently transform contracts written in Peyton Jones contract descriptive language to Solidity," in *2019 19th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka, 2019, pp. 1–8.

[20] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, Jun. 2000.