Smart Contract Synthesis via Multi-modal Specifications

Tanglin Chen
ShanghaiTech University
Shanghai, China
chentl2024@shanghaitech.edu.cn

Haoxian Chen*
ShanghaiTech University
Shanghai, China
hxchen@shanghaitech.edu.cn

Huilin Xiang
ShanghaiTech University
Shanghai, China
xianghl2023@shanghaitech.edu.cn

Yuepeng Wang
Simon Fraser University
Burnaby, Canada
yuepeng@sfu.ca

Abstract

Smart contracts manage substantial digital assets on blockchains. Given their transparency and immutability, ensuring their correctness before deployment is essential. While verification tools offer assistance, the iterative nature of the process presents hurdles, especially for less experienced developers. In this paper, we explore a paradigm shift in smart contract development: writing high-level specifications instead of low-level code. To realize this goal, we propose SmartSpec, a synthesizer that generates smart-contract implementations from functional and safety specifications and validates them with integrated verification. To avoid overly restrictive outcomes, SmartSpec employs maximally permissive synthesis, ensuring contracts preserve all safety requirements while remaining as permissive as possible. Its novel algorithm combines deductive and inductive synthesis techniques, balancing precision and generalization. On 27 contracts, synthesis runs in seconds to minutes; 23 match references without interactive refinement (85%), while the remaining four can be corrected with up to four rounds of interactive feedback through extra examples. Mean gas overhead vs. references on the table is +0.7%.

CCS Concepts

• **Software and its engineering → Software verification and validation**; *Specification languages.*

Keywords

Smart contracts, program synthesis, formal verification, declarative specifications, temporal logic, blockchain

ACM Reference Format:
Tanglin Chen, Haoxian Chen, Huilin Xiang, and Yuepeng Wang. 2026. Smart Contract Synthesis via Multi-modal Specifications. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3837430

*Corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837430

1 Introduction

Smart contracts have gained significant traction in the finance and business sectors, where they manage substantial assets and facilitate high-value transactions. However, these contracts are frequently targeted by attacks, leading to considerable financial losses [12, 23]. As a result, ensuring the correctness of smart contracts is critically important. Although various verification tools have been developed to enhance smart contract security [7, 25, 26, 36, 39], the process often requires multiple iterations between the programmer and the tool to produce high-quality code. Moreover, many smart contract designers lack expertise in formal methods, making it difficult for them to interpret and act on the feedback provided by such tools.

This paper explores a synthesis-oriented approach to smart contract development. Instead of manually implementing and repeatedly verifying code, developers provide formal specifications of the desired contract behaviors, and an automated algorithm generates implementations that satisfy these specifications. This shifts the effort from low-level programming and debugging to high-level specification design, which is more declarative, easier to revise, and more flexible in capturing corner cases directly.

To this end, we study the smart contract synthesis problem. How should smart contracts be specified in a precise yet practical way? Given such specifications, can we automatically generate a provably correct implementation? Addressing these questions requires overcoming several key challenges.

Challenge 1: precise yet practical specification. Synthesizing correct smart contracts requires specifications that are both precise and practical. Input-output examples have proven useful in domains like string manipulation [21, 31], program analysis [32, 33], and query generation [37], but they offer no formal guarantees, which is unacceptable for security-critical contracts. Formal specifications [10, 41] offer strong correctness guarantees but demand expertise and manual effort. Even lightweight alternatives like DeCon [8] fall short for properties involving long-term behaviors [7, 11, 28].

Multiple modalities can be both precise and practical. We observe that combining modalities, more specifically, relational and temporal logic, can cover both local function behaviors and global safety properties over execution histories. We adopt *first-order relational logic* [8] to define contract state through inference over transaction histories, supporting precise reasoning about function-level correctness. We further integrate *temporal logic* [28] to express safety properties across transaction sequences. This unified framework captures both local and global correctness. As shown in

Section 2, such specifications could have prevented real-world logic bugs [6], and our evaluation (Section 6) confirms their effectiveness in guiding correct synthesis across 27 contracts.

Challenge 2: generalizability. Even with precise specifications, the space of implementations remains large. In particular, safety properties often under-constrain program behaviors, making the synthesizer produce overly restrictive contracts that rule out legitimate transaction sequences. Such contracts satisfy the specification but fail to generalize to unseen scenarios.

Trace-Bounded Maximally Permissive Synthesis. To generate safe yet general smart contracts, we adopt *trace-bounded maximally permissive synthesis*: among all correct candidates, we select the implementation that admits the largest number of valid traces in a representative sample set. Since the full trace space is infinite, permissiveness is approximated over a finite sample set rather than optimized in an exact sense; the formal definition and its guarantees are given in Section 5.3. This approximation effectively guides the synthesis toward more general smart contracts.

Challenge 3: efficient synthesis under complex semantics. In general, it is challenging to synthesize a program over a large search space. Moreover, the rich semantics of real-world smart contracts introduce complicated constraints that can easily overwhelm state-of-the-art SMT solvers [7].

Combining deductive synthesis with trace-guided refinement. We address this challenge by dividing synthesis into two phases. Specifically, the first phase performs *deductive synthesis*, which generates a contract sketch using deterministic rules based on the given specifications [8]. The second phase conducts *inductive refinement*, which searches for completions that satisfy a set of concrete traces. To reduce the verification overhead, we precompute the contract state snapshots over these traces and avoid symbolically encoding full transitions. This approach narrows the search space and enables efficient constraint solving, offering a scalable alternative to syntax-guided synthesis [5, 33].

As a testbed, we implemented SmartSpec, a smart contract synthesizer based on a multi-modal specification that consists of: (1) *functional specifications* in the form of inference rules over contract states and transaction records; and (2) *safety specifications* expressed in temporal logic over transaction sequences. Leveraging both deductive and inductive synthesis techniques, SmartSpec generates an abstract representation of the smart contract that satisfies all specified properties while being trace-bounded maximally permissive. This representation enables: (i) efficient verification against specified safety properties; (ii) connections between specification and synthesis output, making synthesized state updates and guards inspectable before deployment. Upon finding a satisfactory contract, its abstract representation is compiled into executable code on the Ethereum [40] platform. SmartSpec currently targets standalone contracts whose logic is expressible in relational logic; supporting cross-contract interactions remains a future work (Section 6.4).

Contributions. Our main contributions are:

- **Multi-modal specification.** We formulate the smart contract synthesis problem using a specification framework that integrates functional specifications (inference rules) and safety specifications (temporal logic) (Section 4).

```
1 mapping(address => Stake) public stakes;
2 struct Stake { uint256 amount; bool unstaked; }
3 function unstake() external {
4     Stake storage s = stakes[msg.sender];
5     require(s.amount > 0); // Missing check:
          require(!s.unstaked);
6     s.unstaked = true;
7     payable(msg.sender).transfer(s.amount);
8 }
```

Figure 1: Simplified staking example vulnerable to repeated withdrawals.

- **Synthesis algorithm.** We propose an efficient algorithm combining deductive and inductive synthesis with a compact encoding of contract semantics, together with soundness and relative completeness guarantees (Section 5).
- **Implementation and evaluation.** We implement SmartSpec and evaluate it on 27 contracts, demonstrating effectiveness in synthesizing verifiably correct smart contracts (Section 6).

2 Background and Motivation

Smart contracts are programs deployed on blockchains to manage digital assets and automatically enforce agreements without intermediaries. They encode core application logic such as token transfers, staking, and auctions, and execute deterministically once deployed. In practice, developers implement smart contracts in domain-specific languages such as Solidity [1] for Ethereum [40], as well as Vyper [2] and Move [41]. In this work, we focus on Solidity contracts due to its widespread adoption.

Vulnerabilities and specifications. We motivate the design of multi-modal specifications using two illustrative smart contract examples inspired by real-world incidents.

Example 1: Repeated withdrawal. Figure 1 shows a contract adapted from a real-world incident [15]. It allows users to stake and later withdraw their tokens. However, the `unstake` function fails to verify if a user has already withdrawn their stake. As a result, an attacker can repeatedly call `unstake` to drain the contract's balance. □

This vulnerability arises from contract-specific logic that is difficult for generic verification tools to capture [6, 7, 26, 28]. Even with a formal specification, existing tools only flag the issue post hoc. In contrast, our goal is to ensure such properties are *enforced by construction*, by automatically synthesizing guards that make the contract both provably correct and transparent in behavior.

Safety specification in temporal logic. The above vulnerability can be effectively captured by the specification that, for all stake index i, transaction `unstake(i)` can happen at most once:

$$\square\,(\neg\,(\bullet(\blacklozenge(\text{unstake}(i)))\wedge\text{unstake}(i)))$$

We follow the standard linear temporal logic syntax in VerX [28], where $\blacklozenge p$ denotes $\text{once}(p)$ and holds if p has occurred at or before the current position, $\bullet p$ denotes $\text{prev}(p)$ and holds if p held at the immediately preceding position (and is false at the initial position), and $\square p$ denotes $\text{always}(p)$. With this specification, we ensure that every stake index can be unstaked at most once, thereby preventing the above attack and enhancing the contract's security.

```
1 function transfer(address sender, address
      recipient, uint256 n) {
2     uint256 s = balance[sender];
3     uint256 r = balance[recipient];
4     require(s >= n);
5     balance[sender] = s-n;
6     balance[recipient] = r+n; // vulnerable
          when sender = recipient
7 }
```

Figure 2: Simplified transfer function vulnerable to inconsistent updates.

In addition to forbidden execution traces, another major class of vulnerabilities arises from incorrect state updates.

Example 2: Balance double-counting. The contract in Figure 2 is adopted from a real-world incident [14]. Suppose an attacker initially holds 20 tokens and transfers 10 tokens to its own address, *i.e.,* the sender and recipient are the same. The contract first reads the sender's balance `s = 20` and the recipient's balance `r = 20`. It then subtracts 10 from `s` and adds 10 to `r`, resulting in a final balance of 30 tokens instead of 20. This inconsistency arises because both updates operate on the same account but are computed independently, violating the intended relational invariant that total tokens must remain constant. Such invariants are naturally expressed as inference rules. □

Functional specification in inference rules. This vulnerability can be prevented by enforcing an invariant stating that an account's `balance` always equals the total tokens received minus the total tokens sent, ensuring that every balance reflects only legitimate transfers. See an example of such specification shortly in Section 3.

These examples demonstrate that both correctly validating incoming transactions and properly updating contract states are critical for maintaining smart contract safety. In the following sections, we show how SmartSpec uses these specifications to synthesize provably correct smart contracts.

3 Overview

Figure 3 shows the architecture of SmartSpec. It takes a multi-modal specification: (1) *inference rules* that define contract states in relational form over past transaction history, and (2) safety properties in *temporal logic* over transaction sequences.

The synthesis proceeds in two phases. First, it deductively transforms the inference rules into a contract sketch that encodes state updates without condition guards. Then, it completes the sketch by inferring transaction guards that ensure all safety properties are satisfied, using an iterative CounterExample-Guided Inductive Synthesis (CEGIS) procedure [34].

Inference rules. Figure 5a presents the inference rules for a crowdfunding contract, whose goal is to raise a target amount within a limited timeframe. Here, contract states are represented as relations, defined by inference rules [8] of the form *head* :- *body*. The rules read from right to left: if all predicates in the body hold, the head predicate is derived. The first rule defines an account's balance as the difference between its total incoming and spent tokens. For instance, Line 1 of Figure 5a sums all investments to compute the total raised. Lines 2–3 track each participant's invested and refunded amounts, and Line 4 derives their balance as the difference.

Step 1. Deductive synthesis (Section 5.1). From the inference rules, SmartSpec synthesizes the contract sketch in Figure 5b, which incrementally maintains these states upon each transaction commit. For instance, Figure 5b contains the update `raised <- raised + n`. It is generated from the rule in Line 1 of Figure 5a, where `raised` is defined as the sum of the `amount` column of the `invest` transactions. Therefore, when a new `invest` transaction is committed, `raised` is incremented by the value `n`.

Safety specification in temporal logic. Figure 6a illustrates three safety properties for the crowdfunding contract, expressed in temporal logic. For example, Line 4 specifies that whenever the crowdfunding is in the `OPEN` state, the sum of all participant balances must equal the `raised` amount. The operator □ (always) enforces that the specified invariant must hold throughout all reachable states. Other operators, such as • (prev, Line 7) and ♦ (once, Line 11), refer to the preceding and any past transaction states, respectively.

Step 2. Counterexample-guided inductive synthesis. The goal of this step (Section 5.2) is to infer transaction guards so that the contract satisfies the safety properties. Because temporal safety often spans multiple transactions, it cannot be encoded by local deductive rules alone. SmartSpec therefore adopts a CEGIS loop. Specifically, SmartSpec starts with empty guards and checks the sketch against the verifier. If there is a safety violation, the verifier returns a counterexample trace, from which SmartSpec infers new guards to block the unsafe behavior. This loop continues until a fully-verified contract is found or it reaches the iteration bound.

Disambiguate via trace-bounded maximal permissiveness (Section 5.3). In practice, even comprehensive safety properties and inference rules rarely capture a contract in full, leaving room for multiple correct implementations. Simple criteria like minimal code size may produce trivial contracts. For instance, a contract that always rejects `unstake` calls is technically safe but defeats its purpose. To avoid this, SmartSpec adopts a trace-bounded maximally permissive strategy. Since exact quantification over the infinite trace space is intractable, SmartSpec approximates permissiveness by sampling legal traces from the safety specification.

As illustrated in Figure 4, the input space includes both safe and unsafe regions: the top-right area denotes counterexamples, while two alternative candidates are safe but differ in permissiveness, and the one admitting more valid traces is preferred.

Scope and limitations. SmartSpec focuses on contracts whose logic can be expressed as inference rules with deterministic guards and fixed, side-effect-free user-defined functions (UDFs), and does not handle external interactions (*e.g.,* oracle calls or cross-contract calls). Safety guarantees follow the verifier (unbounded with inductive invariants, otherwise bounded), and trace-bounded maximal permissiveness is defined with respect to sampled traces, thus approximating the full behavior space.

4 Problem Formulation

This section formalizes the synthesis problem. We begin by defining the syntax of inference rules and safety properties, then specify the output program space.

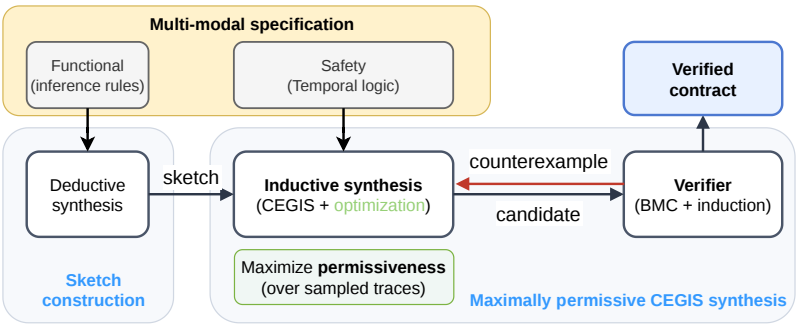

Figure 3: Overview of SmartSpec.

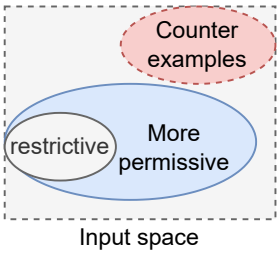

Figure 4: Permissiveness.

```
raised(s) :- s = sum m: invest(_, m).
investTotal(p,s) :- invest(p, _), s = sum m:
    invest(p,m).
refundTotal(p,s) :- refund(p, _), s = sum m:
    refund(p,m).
balanceOf(p,s) :- investTotal(p,i), refundTotal
    (p,r), s := i-r.
```

(a) Specification: inference rules over transaction records.

```
invest(p: address, n: uint256): {
    raised <- raised + n;
    balanceOf[p] <- balance[p] + n;
} // other transactions ...
```

(b) Partial program generated from inference rules, incrementally maintaining states on new transaction commits.

Figure 5: Phase 1: deductive synthesis.

```
// P1: In the OPEN state, balances sum to the
    raised amount
.decl sumOfBalance(n: uint)
sumOfBalance(s) :- s = sum n: balanceOf(_,n).
□(state(s) ∧ sumOfBalance(m) ∧ raised(n) ∧ s==
    OPEN
    ⟹ m==n);

// P2: Refund subtracts balance from raised
    amount
□(refund(p,n) ∧ •(balanceOf(p,n')) ∧ raised(m)
    ∧ •(raised(m'))
    ⟹ m==m'-n');

// P3 Withdraw and refund cannot be both
    executed
□(¬(⧫(withdraw) ∧ ⧫(refund)));
```

(a) Specification in linear temporal logic.

```
invest(p,n)  -: msgSender(p),msgValue(n),closed
    (false),raised(s),target(t),s<t.
closed(true) -: msgSender(s), owner(s).
refund(p,n)  -: msgSender(p),closed(true),
    raised(r),target(t),r<t, balanceOf(p,n),n
    >0.
withdraw(p,r)-: msgSender(p), beneficiary(p),
    raised(r),target(t),r>=t.
```

(b) Synthesis intermediary output: transaction validation rules.

Figure 6: Phase 2: inductive synthesis.

Semantics summary. A *transaction type* $Tx(x_1, \ldots, x_n)$ is a named, parametrized operation (*e.g.*, `transfer(from, to, amount)`). By default a transaction type may occur any number of times in a trace; the temporal specification restricts frequency when needed (*e.g.*, $\square\,(\neg\,(\bullet(\blacklozenge(\mathit{unstake}(i))) \wedge \mathit{unstake}(i)))$ to ensure each stake index is unstaked at most once). Specifications apply to individual contract *instances*, not to all deployed instances globally.

Functional specification. We start by introducing relations, the basic building blocks of contract states. We then define predicates constructed over these relations, and finally describe how rules are formed from predicates to specify contract functionality.

Relations. Contract states are modeled as relations declared by the user (Figure 7a) and defined through inference rules. We support three kinds of declared relations: (1) *Singleton relations*, which contain exactly one tuple and represent global configuration parameters (*e.g.*, funding target or deadline); (2) *Transaction relations*, which record transaction invocations such as `invest` or `refund`; and (3) *Simple relations*, which store persistent contract state and may specify primary key columns `[i1,i2,...]` to uniquely identify tuples (*e.g.*, `balanceOf(addr, amount)` keyed by address). Platform-provided primitives (*e.g.*, `msg.sender`, `timestamp`, `send`) are built-ins and need not be declared.

Predicates. As shown in Figure 7b, predicates (*Pred*) are the logical atoms used in rule bodies and transaction guards. They include: (1) relational atoms $R(\bar{x})$ over declared relations, (2) comparisons $C(\bar{x})$ between domain-consistent values, (3) arithmetic function applications $y = F(\bar{x})$, and (4) calls to user-defined functions (UDFs).

UDFs are optional, side-effect-free predicates written as $y = \mathit{id}(\bar{x})$ (Figure 7b): standard Solidity helpers for fragments that are clumsy as pure joins, lowered and evaluated on traces like other predicates. For instance, the Uniswap V2 0.3%-fee formula [3]

$$\mathit{amountOut}(a_{\text{in}}, r_{\text{in}}, r_{\text{out}}) = \frac{a_{\text{in}} \cdot 997 \cdot r_{\text{out}}}{r_{\text{in}} \cdot 1000 + a_{\text{in}} \cdot 997}$$

can be named once (*e.g.*, as `amountOut`) and reused in guards with relational constraints (*e.g.*, a slippage floor); state change and temporal properties remain in the rules.

Rules. SmartSpec uses Horn-clause style inference rules, a subset of first-order relational logic. As shown in Figure 7b, each rule has the form $\mathit{Head}(\bar{x})$:- Body, read right to left: if the body holds, the head tuple is derived and inserted into its relation.

$$
\begin{array}{rcl}
Decl & ::= & RelDecl^* \\
RelDecl & ::= & SingletonR(\bar{T}) \mid Tx\ R(\bar{T}) \mid Rel\ R(\bar{T})\ [Key] \\
Key & ::= & [i_1, i_2, \ldots] \\
T & ::= & \texttt{int} \mid \texttt{addr} \mid \texttt{bool} \mid \ldots
\end{array}
$$

(a) Relation declarations.

$$
\begin{array}{rcl}
Rule & ::= & H(\bar{x}) :\text{-}\ Body \\
Body & ::= & Join \mid R(\bar{x}), y = Agg : R(\bar{y}) \\
Join & ::= & R(\bar{x}) \mid Pred, Join \\
Agg & ::= & sum\ n \mid max\ n \mid min\ n \mid count \\
Pred & ::= & R(\bar{x}) \mid C(\bar{x}) \mid y = F(\bar{x}) \mid y = id(\bar{x}) \\
C & ::= & > \mid < \mid \geq \mid \leq \mid \neq \mid == \\
F & ::= & + \mid - \mid \times \mid /
\end{array}
$$

(b) Inference rules.

$$
\begin{array}{rcl}
F & ::= & atomic \mid u\_op\ F \mid F\ bin\_op\ F \\
atomic & ::= & R(\bar{x}) \mid C(\bar{x}) \\
u\_op & ::= & \neg \mid \blacklozenge(\text{once}) \mid \Box(\text{always}) \mid \bullet(\text{prev}) \\
bin\_op & ::= & \wedge \mid \vee \mid \rightarrow
\end{array}
$$

(c) Temporal logic for safety specification.

Figure 7: SmartSpec **specification syntax.**

We distinguish two rule types:

(1) Join rules. The body of a rule may be a conjunction of predicates, forming a `Join` rule. Such a rule holds if there exists a variable assignment $\pi : V \mapsto D$ that makes all literals true. Relational literals are satisfied when the corresponding tuple exists in the relation, while conditions and functions impose constraints on variables. For example, the rule at Line 4 in Figure 5a joins `investTotal(p,i)` and `refundTotal(p,r)` on the participant p, and derives `balanceOf(p,s)` with $s = i - r$.

(2) Aggregation rules. These take the form $y = Agg\ n : R(\bar{y})$, which computes an aggregate over all rows in $R(\bar{y})$ matching a given variable assignment.

```
investTotal(p,s):-invest(p,_),s=sum m:invest(p,m).
```

This rule groups the `invest` table by the first column (`p`) and, for each group, sums the second column (`m`) to produce `investTotal`.

A rule is *well-formed* if all variables are grounded, *i.e.,* each variable must appear either as a transaction parameter or as a bound variable in a relational atom. For example, in `transfer(sender, recipient,amount)`, predicates such as `amount > 0` or `balanceOf( sender,b), b >= amount` are well-formed, whereas `x > 0` is not if `x` appears nowhere else.

Safety specification. Figure 7c shows the syntax of past time linear temporal logic [28]. Atomic propositions represent relations $R(\bar{x})$ and conditions $C(\bar{x})$, whose syntax is defined in Figure 7b. Temporal operators include once, always, and prev. In addition, formulas can be constructed through logical connectives including conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$). There are two typical approaches to specify safety properties:

(a) Patterns over transactions. These are constructed using temporal and logical operators over transactions. For instance:

$$\Box(\neg(\blacklozenge(\texttt{withdraw}) \wedge \blacklozenge(\texttt{refund})))$$

Table 1: Fragments of Solidity supported by SmartSpec**.**

Feature	Restriction
ETH transfer	Supported.
Loops	Bounded loops.
Recursion	No recursion.
External calls	No calls to external contracts.
Contract	No contract creation or destruction within a transaction.

where `withdraw` and `refund` represent two types of transactions. This formula indicates that once a crowdfunding beneficiary has withdrawn funds, no investor can request a refund, and vice versa.

(b) Patterns over contract states. Relational predicates $R(\bar{x})$ are instantiated on relations representing contract states. For instance:

$$\Box(\texttt{state}(s) \wedge \texttt{sumOfBalance}(m) \wedge \texttt{raised}(n) \wedge s = \text{OPEN} \rightarrow m = n)$$

This formula specifies an invariant over contract states, represented by relations `state`, `sumOfBalance`, and `raised`. This invariant states that, while the crowdfunding is in the `OPEN` state, the sum of all participant balances must equal the raised amount.

Output space. SmartSpec targets a subset of Solidity, using DeCon [8] as an intermediate representation. As shown in Table 1, this subset supports ETH transfers and bounded loops over arrays, but excludes recursion, external calls, and contract creation or destruction, since the verifier [7] cannot handle them. Recursion is disallowed in the specification to ensure termination and is uncommon in practice [8]. Extending SmartSpec to reason about external interactions remains important future work. Despite these limitations, 44% of the top-50 Ethereum contracts by transaction volume fall within this fragment directly, rising to 76% with a lightweight `assume`-block extension for cross-contract calls (Section 6.4).

Correctness criterion. A synthesized contract C is considered *correct* if: (1) all temporal safety properties are formally verified by DCV [7], using inductive proofs with fallback to bounded model checking; and (2) C is manually inspected and tested against the reference source to confirm semantic consistency with the intended behavior. Correctness is semantic, not syntactic: two contracts are equivalent if they satisfy the same relational and temporal specifications. The second step catches errors from ambiguous or under-constrained specs, *e.g.,* a guard that picks < instead of <= on boundary traces that were absent from the sample set (Section 6).

5 Synthesis Algorithm

This section presents the deductive and inductive synthesis algorithm, and synthesis output disambiguation via maximal permissiveness. We build on existing verification infrastructure [7]: bounded model checking for counterexamples, induction for proofs, and declarative structure for efficient invariant search; full details are provided in the full version [9].

5.1 Deductive Synthesis

In the deductive synthesis phase, SmartSpec repurposes the transformation rules of DeCon [8], a declarative language for smart contracts, to generate executable code from inference rules. Unlike

Algorithm 1 CEGIS Loop with Maximal Permissiveness.

Input: (1) Φ: safety specification; (2) $\mathcal{R}$: functional rules; and (3) K: iteration bound (hyperparameter).
Output: A synthesized contract C or $\perp$ if not found.

1: $\mathcal{T}_{\text{dis}} \leftarrow$ SampleValidTraces(Φ) ▷ Disambiguation (Sec. 5.3).
2: $\mathcal{P} \leftarrow$ EnumeratePredicates($\mathcal{R}$)
3: $\mathcal{T}_{\text{cex}} \leftarrow \emptyset$
4: $C \leftarrow$ InitSketch($\mathcal{R}$) ▷ Section 5.1.
5: $iteration \leftarrow 0$
6: **while** $iteration < K$ **do**
7: $iteration \leftarrow iteration + 1$
8: $\tau \leftarrow \text{BMC}(C, \Phi)$ ▷ Find counterexample trace
9: **if** $\tau = \perp$ **then**
10: **return** C ▷ No counterexample, synthesis complete
11: **else**
12: $\mathcal{T}_{\text{cex}} \leftarrow \mathcal{T}_{\text{cex}} \cup \{\tau\}$
13: **end if**
14: $C \leftarrow$ BlockTraceConstraints($\mathcal{T}_{\text{cex}}, \mathcal{P}$) ▷ Section 5.2.
15: $J \leftarrow$ PermissivenessObjective ($\mathcal{T}_{\text{dis}}, C$) ▷ Section 5.3.
16: $C \leftarrow$ Optimize(C, J) ▷ Z3: satisfy constraints, maximize permissiveness
17: **end while**
18: **return** $\perp$ ▷ Iteration bound reached

DeCon, which requires developers to manually specify conditions, SmartSpec focuses only on generating state maintenance logics, delegating condition generation to an automated synthesizer.

For each transaction type, the contract sketch consists of the state maintenance instructions for newly committed transactions. The deductive synthesis begins by identifying dependencies among declared relations. A relation R depends on Tx if there exists a rule where R is in the head and Tx in the body. Starting with the relation for Tx, update instructions are derived to ensure the validity of the inference rule results as new relations are derived. The dependency relation is transitive: updates to a relation trigger further updates to other affected relations. This process iterates through the dependency chain until no further updates are needed. Since recursive relations are prohibited in the specification (Section 3), this process is guaranteed to terminate.

Example 3: Figure 5b shows the contract sketch for the crowdfunding example. The deductive phase produces update instructions along dependency chains. In this example, instructions are generated following two dependency chains: `invest` → `raised`, and `invest` → `investTotal` → `balanceOf`. Note that `investTotal` is eliminated by the compiler optimizer that short-circuits updates from `invest` to `balanceOf`. □

5.2 Inductive Synthesis

Building on the contract sketch generated above, SmartSpec refines it by inferring transaction guards that ensure safety. The goal is to rule out all counterexample traces produced by bounded model checking while avoiding overly restrictive conditions. To this end, SmartSpec adopts a constraint-based, counterexample-guided inductive synthesis (CEGIS) procedure enhanced with a trace-bounded maximal permissiveness objective, which favors solutions that are as general as possible. The rationale and formulation of this objective are detailed in Section 5.3.

Overview. Algorithm 1 outlines the CEGIS loop. Given functional rules $\mathcal{R}$ and safety specification Φ, SmartSpec first samples a set of disambiguating traces $\mathcal{T}_{\text{dis}}$ that satisfy Φ (Section 5.3) and constructs the predicate set $\mathcal{P}$ by enumerating type-consistent bindings of relational variables and comparison operators (Figure 7b) for guard synthesis; an optional optimization restricts the binding space using domain conventions (Section 5.2). Next, a contract sketch C is initialized from $\mathcal{R}$ (Section 5.1), containing state updates but no guards.

In each iteration, bounded model checking (BMC) verifies C against Φ. If no counterexample is found, synthesis terminates. Otherwise, the counterexample τ is added to $\mathcal{T}_{\text{cex}}$, and SmartSpec constructs blocking constraints $\mathcal{C}$ from $\mathcal{T}_{\text{cex}}$ and $\mathcal{P}$ (Section 5.2) and defines a permissiveness objective J that measures the number of disambiguating traces accepted by C. The solver Z3 optimizes C to satisfy $\mathcal{C}$ while maximizing J, repeating until no counterexample is found or the iteration bound K is reached.

Predicate generation. In line 2, for each transaction type Tx, SmartSpec constructs a predicate set $P(Tx)$ used in guard synthesis; $\mathcal{P}$ aggregates these sets. Predicates are formed by instantiating domain-consistent comparison operators (C) of Figure 7b. Variables in a predicate may originate from either the transaction parameters or contract states represented as relations.

For example, consider transaction `transfer` with parameters `sender`, `recipient`, and `amount`. Predicates may include parameter-based conditions such as `amount > 0`, or state-dependent ones such as `balance(sender)>= amount`, where variables are bound to relational states.

A predicate is considered *well-formed* if all of its variables are grounded, *i.e.,* each variable must appear either as a transaction parameter or as a bound variable in a relational tuple. For example, in `transfer(sender, recipient, amount)`, the predicate `balance(sender)>= amount` is well-formed because both `sender` and `amount` are grounded. In contrast, `balance(p)>= amount` is ill-formed if `p` is not bound to any parameter or relational variable, since its value cannot be resolved during evaluation. This ensures that all predicates can be evaluated concretely over the current contract state and transaction inputs.

Optimization via parameter-pattern restriction. Full enumeration yields a Cartesian explosion and incorrect bindings. We apply simple heuristics (*e.g.,* distinct roles bind to distinct parameters; non-zero for amounts and addresses) and restrict to these patterns first, falling back to the full space if UNSAT. For `transferFrom`, the binding `allowance(owner, spender, m)` is correct. Binding the first argument to `recipient` is not. Completeness is preserved: Proposition 5 holds regardless of which search phase succeeds.

With the predicate set $\mathcal{P}$ in place, we next describe the contract encoding and constraint generation process (Algorithm 1, line 12).

Internal contract representation. Building on the predicate set generated above, SmartSpec encodes each candidate transaction guard into a constraint representation suitable for solver optimization. As shown in Figure 8, each conjunctive clause is represented

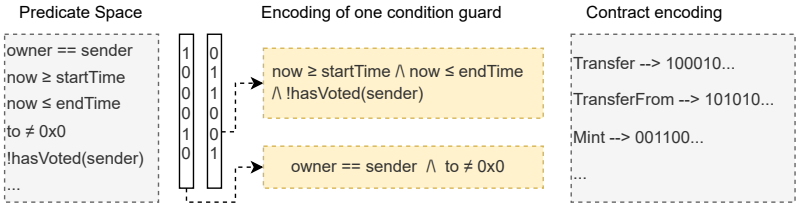

Figure 8: Encoding of a candidate contract

Table 2: Truth table for an example trace.

Transactions	ended	sender==owner	n>0	...
invest(a1,10)	F	F	T	
invest(a2,50)	F	F	T	
invest(a3,60)	F	F	T	
end()	T	T	NA	
withdraw(a1)	T	F	NA	

as a one-hot vector over the predicate space. For example, the vector `100010...` corresponds to the conjunction of the first and fifth predicates, `owner == sender` ∧ `to` ≠ `0x0`. This encoding bridges the high-level logical specification and the bit-vector formulation used by the solver in Algorithm 1.

The entire contract is represented as a mapping $\Phi : Tx \rightarrow \text{List}[Boolean]$, where each transaction type Tx is associated with a Boolean vector of length $|P|$ (the size of the predicate set), indicating the inclusion of each predicate in its guard.

Constraints on example traces. To ensure correctness, we require that viable candidate contracts reject all counterexamples. Rather than symbolically encode the full contract semantics, we simplify constraint generation by evaluating predicate truth values on concrete traces and collecting the results in a lookup table $\mathcal{D}$ over which the constraints are defined.

(a) Evaluate predicate truth value. Using the contract sketch, which specifies how state updates occur upon transaction commits, we simulate the execution of concrete example traces to precompute the contract state at each step. This allows us to directly evaluate the truth value of each predicate $P(Tx)$ for every transaction Tx in the trace, bypassing symbolic execution and simplifying the encoding.

Example 4: Table 2 shows predicate truth values for an example trace. Each row corresponds to the state after committing a transaction (header column), with predicate values computed from the evolving contract state based on the partial sketch. For instance, `ended` is initially false and becomes true after the `end` transaction. Predicates irrelevant to a transaction type (*e.g.,* `n > 0` for `end` or `withdraw`) are marked `NA`. □

(b) Encoding example traces as constraints. Each example trace is encoded as constraints over its predicate truth table: a transaction is approved if all predicates in its guard are satisfied. Formally, let Γ_T be the truth table for trace T:

$$approve(tx, \Phi) \triangleq \forall i.\ \Phi[i] \rightarrow \Gamma_T[tx][i]$$

Here, $\Phi[i]$ indicates whether the i-th predicate is included in the guard of transaction tx. The implication enforces that if a predicate is selected in the guard, it must evaluate to true for the transaction in the given trace.

A trace is considered valid only if all its transactions are approved:

$$approve(T, \Phi) \triangleq \forall tx \in T.\ approve(tx, \Phi)$$

Intuitively, this means that for a trace to be accepted, every transaction within it must satisfy all predicates selected in its guard.

For a counterexample trace set Neg, the constraint is:

$$\{\neg approve(T, \Phi) \mid T \in \text{Neg}\}$$

Together, these encodings define the constraint space used by the inductive synthesizer. Each counterexample trace contributes a blocking constraint that eliminates unsafe behaviors, while satisfying assignments to Φ represent candidate guard conditions consistent with all collected examples. In the subsequent optimization phase, SmartSpec searches for the most permissive solution that satisfies these constraints, as described in Section 5.3.

5.3 Trace-Bounded Maximally Permissive Synthesis

As discussed in Section 3, the admissible behaviors of a smart contract are defined by temporal safety properties. However, contracts that reject most or all transactions, though safe, are useless. To avoid such outcomes, SmartSpec adopts a *trace-bounded maximally permissive* synthesis strategy, aiming to accept as many valid traces as possible without violating safety properties.

Since the trace space is infinite, exact quantification is infeasible. We instead sample a subset of valid traces and maximize the number accepted. A small but diverse sample is often sufficient to distinguish generalizable contracts from overly restrictive ones.

Trace sampling. We construct a small set of up to N blockchain addresses and randomly initialize parameters such as the contract owner and permitted users. Setup transactions are executed to populate account states, since many checks depend on balances or role assignments (*e.g.,* owner, operator). For each transaction type, we randomly draw type-consistent inputs, execute them to generate traces, and retain only those satisfying safety properties. For efficiency, a fixed number of traces (N per transaction type) is sampled.

The permissiveness of each candidate contract is encoded as an optimization objective in the Z3 solver used during the inductive phase. The solver thus searches for contracts that block all counterexample traces while accepting as many valid traces as possible.

Example 5:[Permissiveness disambiguation] Consider a staking contract with the safety property that each stake index i can be unstaked at most once. Suppose the sampled valid trace set contains:

$$T_1 : \langle stake(a, 10, 7),\ advance(8),\ unstake(a, i)\rangle$$
$$T_2 : \langle stake(b, 20, 30),\ advance(31),\ unstake(b, j)\rangle$$
$$T_3 : \langle stake(a, 5, 7),\ advance(8),\ unstake(a, k)\rangle$$

and a counterexample trace:

$$T_\times : \langle stake(a, 10, 7),\ advance(8),\ unstake(a, i),\ unstake(a, i)\rangle.$$

Consider two candidate guards for `unstake(u,i)`:

C_1 : `lockExpired`$(i) \wedge \neg$ `unstaked`$(i) \wedge$ (`msg.sender` = `owner`)

C_2 : `lockExpired`$(i) \wedge \neg$ `unstaked`(i)

Both candidates block $T_\times$ and satisfy the safety property. However, C_1 rejects T_1, T_2, T_3 unless the owner performs every withdrawal, while C_2 accepts all three valid traces by allowing the stake owner to withdraw after expiry. Hence C_2 is more permissive on the sampled traces and is preferred by SmartSpec. □

Remarks. Trace-bounded maximal permissiveness is guaranteed only with respect to the generated traces; more permissive correct implementations may exist if the trace set is biased. In practice (Section 6), we find that 50 traces per transaction type suffice to avoid such cases.

Guard expressiveness and tradeoffs. The current synthesis targets guards that are conjunctions of relational predicates (Section 5.2). This restriction is intentional: (1) virtually all `require` guards in our benchmarks are conjunctions of simple predicates (*e.g.,* `amount > 0 && balance[sender] >= amount`), so the fragment is practically complete for this domain; (2) conjunctions are cheap to evaluate both at synthesis time (bit-vector encoding) and at runtime (short-circuit evaluation). This restriction is not fundamental: disjunctions can be supported by encoding the guard as a disjunction of conjunctive clauses (DNF), with the clause count as a tunable parameter controlling expressiveness vs. solving cost. Complex subexpressions can be wrapped as user-defined functions (UDFs) and treated as atomic Boolean candidates. Their truth values are precomputed on each trace point so the synthesis engine sees a plain Boolean (*e.g.,* the Uniswap V2 fee formula is already handled this way).

5.4 Properties

We formalize the correctness guarantees of SmartSpec. Specifically, we establish (1) soundness with respect to the functional and safety specifications, and (2) relative completeness within the defined search space.

We first state a lemma about the verifier, which forms the foundation of SmartSpec's soundness.

Lemma 1 (Verifier guarantee): *Let Φ be a candidate contract.*

(1) If the verifier proves Φ as an inductive invariant, then Φ holds for traces of arbitrary length.

(2) If induction fails but no counterexample of length $\leq B$ is found, then Φ holds for all traces of length at most B. □

Proofs are deferred to the full version [9].

Proposition 2 (Soundness of inductive phase): *Let C_0 be a contract sketch and Φ a safety specification. If the inductive phase of* SmartSpec *returns a contract C, then C satisfies Φ under the guarantees provided by the verifier (Lemma 1).* □

Proof sketch. SmartSpec returns a contract C only if the verifier reports no violation of Φ. By Lemma 1, this implies that C satisfies Φ under the verifier's guarantees. □

Lemma 3 (Soundness of deductive phase): *Let $\mathcal{R}$ be a well-formed set of functional inference rules. Let $C_0 \leftarrow \textsc{InitSketch}(\mathcal{R})$ be the contract sketch generated by the deductive phase. Then for every transaction sequence T, the states produced by executing C_0 correspond to the tuples derivable from $\mathcal{R}$ under its relational semantics.* □

The above lemmas establish the soundness of SmartSpec.

Proposition 4 (Soundness): *Let $\mathcal{R}$ be the functional specification, Φ the safety specification, and T the sampled trace set used to approximate permissiveness. If* SmartSpec *returns a contract C, then:*

(1) C satisfies the functional specification $\mathcal{R}$;

(2) C satisfies the safety specification Φ;

(3) among all contracts satisfying (1) and (2) within the search space, C maximizes the number of traces in T that are admitted (trace-bounded maximal permissiveness). □

In other words, the deductive phase enforces the intended state semantics, while the inductive phase synthesizes guards that ensure safety. Among all safe completions of the sketch, SmartSpec returns one that is trace-bounded maximally permissive with respect to T.

Proof sketch. By Lemma 3, the sketch preserves the functional semantics of $\mathcal{R}$, and the inductive phase only adds guards without modifying state updates, so the final contract C still satisfies $\mathcal{R}$. Since SmartSpec returns C only after the verifier reports no violation of Φ, Lemma 1 ensures that C satisfies Φ under the verifier guarantee. Moreover, as permissiveness over the sampled trace set T is encoded as an optimization objective, the solver returns a model maximizing it among all satisfying assignments. Hence C satisfies $\mathcal{R}$ and Φ and is maximally permissive with respect to T within the search space. □

Definition 1: Guard search space. Let P be the set of predicates generated from declared relations, domain-consistent comparisons, arithmetic expressions, and user-defined functions (UDFs); SmartSpec may first search a domain-convention-restricted subset and fall back to the full set if the restricted search fails (Section 5.2).

The guard search space $\mathcal{H}$ consists of all guards that can be constructed as conjunctions of predicates from P. □

Proposition 5 (Relative completeness): *Let $\mathcal{R}$ be the functional specification and Φ the safety specification. Let $\mathcal{H}$ be the guard search space defined in Definition 1.*

If there exists a contract C^* obtained by instantiating the sketch with guards from $\mathcal{H}$ such that C^* satisfies Φ, then SmartSpec *will return some contract $C \in \mathcal{H}$ that satisfies Φ.* □

Proof sketch. The inductive phase searches the finite hypothesis space $\mathcal{H}$ via constraint solving. Assuming the underlying solver is complete for the encoded constraint theory, if a satisfying guard assignment exists within $\mathcal{H}$, the solver can discover one. Hence SmartSpec returns such a contract after successful verification. □

Proposition 6 (Complexity): *Let $|\mathcal{P}|$ be the number of predicates, m the number of counterexample traces, and L the maximum trace length. In each CEGIS iteration,* SmartSpec *constructs constraints of size $O(m \cdot L \cdot |\mathcal{P}|)$ and invokes the SMT solver once. Thus, the encoding cost per iteration is polynomial in m, L, and $|\mathcal{P}|$, and the total runtime scales linearly with the number of iterations.* □

The dominant cost of SmartSpec lies in the CEGIS loop. Each iteration involves: (1) bounded model checking to search for counterexamples, and (2) solving a constraint optimization problem whose size is proportional to the number of predicates and accumulated traces. In practice, BMC is the main time consumer, since it explores execution paths over the contract state space, while constraint solving operates over a compact bit-vector encoding

of predicates. Predicate truth evaluation and trace simulation are comparatively lightweight and parallelizable across traces.

This explains two empirical observations: (i) synthesis time grows with the number of relations and transaction types, because they enlarge the predicate space and the transition structure explored by BMC; (ii) per-iteration time remains relatively stable across benchmarks, since the solver operates over similarly structured constraint templates. Thus, performance is primarily governed by the complexity of the state space exposed to the verifier, rather than by the inductive encoding itself.

6 Evaluation

We aim to answer the following research questions:

(1) **Effectiveness and efficiency**. Can SmartSpec synthesize different contracts correctly and efficiently?
(2) **Output quality**. Can SmartSpec generate efficient smart contracts compared to their hand-written counterparts?
(3) **Improvement over baseline tools**. What are the differences and advantages of SmartSpec compared to existing tools?
(4) **Coverage**. How do SmartSpec's syntactic constraints affect practical adoption?

Benchmarks. We evaluate SmartSpec on 27 smart contracts and their temporal specifications from prior work on smart contract verification [7, 28]. We include only contracts that satisfy two criteria: (1) no essential cross-contract interactions (calls, proxies, or delegates), because SmartSpec synthesizes standalone single contracts; and (2) core logic expressible as relational logic. Excluded are, for instance, contracts using recursive data structures, Merkle trees, or zero-knowledge proof verification.

Specification derivation. Specifications come from two sources. (1) Directly taken from prior work, including temporal specifications [7, 28], and relational specification [8]; (2) To increase the benchmark sizes, for contracts missing specification, and additional contracts on Ethereum [40], we manually and systematically derive temporal and relational specification from the source code and ERC standard documentation when available. For example, ERC-20 balances are expressed relationally over transfers:

```
balanceOf(p, n) :- totalIncome(p, m), totalSpent(p
    , o), n = m - o.
```

This derivation is semi-automated: relational specs are written manually; SmartSpec synthesizes Solidity from them; and the output is validated against the reference via inspection and testing.

Table 3 summarizes reference Solidity size (LoC), specification size (#rel., #tx., #rules, #prop.), synthesis run time, CEGIS iterations, gas comparison, and concrete vs. symbolic encoding columns. Full benchmarks are in the artifact [9].

Experiment setup. SmartSpec is implemented in $2k$ lines of Scala, extending DeCon [8] for deductive synthesis and Solidity support, integrating DCV [7] for temporal verification, and using Z3 [13] v4.8.14 for constraint solving. Bounded model checking explores up to 5 transactions per search. For permissiveness analysis, we sample 50 traces per transaction type. We set a one-hour timeout. All experiments run on a 4 GHz CPU with 24 GB memory. End-to-end run times in Table 3 include bounded model checking. We report mean gas per contract from deployment plus a fixed script of function calls, measured on a local EVM.

6.1 Effectiveness and Efficiency

Effectiveness. Table 3 lists 27 benchmarks. The footer shows 23 (85%) with a plain ✓: judged equivalent to the reference without interactive annotation. Four rows carry ✓(k), where k is the number of interactive refinement rounds (see discussion shortly) before we accepted the result. Every row still passes the automated verifier on the stated specification.

Failure cases. Failures arise mainly from SmartSpec's trace-based permissiveness approximation. Scores reflect whether predicates happen to be true or false on a small, synthetically generated trace set; if generation is biased (*e.g.,* the same address pool for roles and users, or uniform setup funding), those scores can skew the permissiveness objective (Section 5.3). The learner has no causal account of why a predicate should act as a guard.

Handling failure via lightweight interactions. To address this limitation, we introduce a lightweight interactive loop. If the designer finds the synthesized contract too restrictive, they supply a concrete positive transaction that ought to be accepted. Because permissiveness is judged from sampled traces, we merge those inputs with the existing trace pool: every candidate guard must accept them, which is enforced directly in the synthesis encoding as additional hard constraints. In our experience, each contract that needed an interactive pass converged in at most four rounds of interactive feedback through extra examples. This shows that SmartSpec remains practically useful even when one-shot synthesis falls short: correction requires only a few concrete examples, not knowledge of the synthesis internals.

Efficiency. Table 3 reports about 2 s–203 s run time per contract (medians 23 s and 25 CEGIS rounds; footer averages 52.8 s and 45.4 rounds). The slowest cases are cappedCrowdSale, erc20, and matic, with 117–120 CEGIS rounds each. The ratio of mean run time to mean rounds is ≈ 1.2 s per cycle, consistent with solver and verifier work each round. Larger #rel. and #tx. enlarge the sketch and predicate space (Section 5.2), *e.g.,* cappedCrowdSale peaks at 41 relations, and need more refinement.

Overhead analysis. Bounded model checking is roughly a third of run time; the rest is the CEGIS loop, which dominates when iterations reach the mid-sixties. One-hot guard solving stays cheap, so harder temporal proofs cost more in verification than in SAT. All runs finish within about two minutes, which is acceptable for medium-sized contracts.

Safety verification. Safety properties over shared state remain the most iteration-hungry, because many transaction types touch the same relations and each guard must preserve invariants. For example, in tether, non-negative balances and non-negative allowances are enforced across mint, transfer, and allowance paths; that benchmark uses 83 refinement rounds, aligning with the need to coordinate guards on every entry point that updates either mapping.

Optimization effectiveness. A key design in the inductive phase is to precompute predicate truth values on concrete traces rather than encoding contract state transitions symbolically in the SMT formula. We quantify the benefit through an ablation study: the baseline uses a standard symbolic execution approach where state updates are encoded as SMT variables and predicates are evaluated

Table 3: 27 benchmarks: LoC of spec, #relations/transactions/inference rules/safety properties, synthesis run time, CEGIS iterations, ablation run time and speedup, mean Ganache execution gas (K) and % change vs. hand-written reference. Result: ✓ = one-shot success (first pass); ✓(n) = succeeded after n additional interactive feedback.

Benchmark	LoC	Specification				Success	Run time				Gas Efficiency	
		#rel.	#tx.	#rules	#prop.	Result	**run (s)**	#iter.	Abl. run (s)	speedup	Gas(K)	Diff
wallet	38	16	3	9	3	✓	7	11	8	1.1×	37	+10.9%
erc20	66	28	5	16	5	✓	129	120	483	3.7×	30	−7.7%
bnb	85	38	8	19	5	✓	98	100	207	2.1×	32	−1.4%
matic	88	40	11	19	7	✓	151	117	> 1200	> 7.9×	36	+2.9%
controllable	82	35	8	19	6	✓	144	113	> 1200	> 8.3×	35	+4.3%
tether	75	33	7	16	5	✓	91	83	413	4.5×	25	+1.7%
brickBlockToken	95	39	9	23	8	✓	6	4	12	2.0×	31	−6.0%
shib	66	29	7	17	4	✓	88	88	210	2.4×	33	+4.0%
tokenPartition	53	21	3	13	4	✓(2)	22	16	96	4.4×	33	−1.5%
wbtc	98	38	11	20	17	✓	41	33	57	1.4×	30	+0.3%
linktoken	77	31	6	20	7	✓	123	112	186	1.5×	32	−6.8%
ltcSwapAsset	77	23	9	19	17	✓(4)	10	2	2	0.2×	45	+13.6%
cappedCrowdSale	95	41	8	21	8	✓	203	120	> 1200	> 5.9×	32	−4.4%
auction	51	20	3	8	12	✓(2)	2	2	4	2.0×	32	+9.7%
crowFunding	68	25	4	17	9	✓	6	9	2	0.3×	32	+2.6%
finalizableCrowdSale	77	38	8	17	6	✓	110	94	298	2.7×	34	−3.8%
voting	40	15	2	9	4	✓	3	4	10	3.3×	30	−6.4%
nft	50	19	5	8	7	✓(4)	3	6	> 1200	> 400.0×	31	+0.7%
sushi	55	24	4	12	4	✓	22	25	21	1.0×	36	+2.2%
erc20roles	73	31	6	16	6	✓	26	28	74	2.8×	34	−4.2%
dai	73	31	6	16	6	✓	23	25	45	2.0×	34	−4.2%
weth	67	27	5	13	10	✓	10	10	6	0.6×	34	−1.3%
stakingRewards	46	19	3	10	6	✓	3	3	1	0.3×	34	+6.4%
comp	84	32	4	29	7	✓	20	18	11	0.6×	35	+4.9%
erc20burnable	73	31	6	16	7	✓	51	52	58	1.1×	36	+2.3%
erc20capped	98	37	5	35	8	✓	23	24	> 1200	> 52.2×	35	+6.8%
uni	90	34	5	29	9	✓	11	6	31	2.8×	33	−7.2%
Avg. / total	72					23/27 (85%) one-shot	52.8	45.4	> 305.0	> 2.5×	33	+0.7%

symbolically during constraint solving (Column Abl. run in Table 3). The concrete encoding reduces synthesis time by >2.5× on average (excluding 5 benchmark runs where baseline exceeds the 20-minute timeout), confirming the effectiveness of this optimization.

6.2 Output Quality

We compare synthesized contracts with hand-written references from open repositories using gas consumption [40], a proxy for user fees and contract efficiency. For Table 3, we use the same deployment and scripted-call procedure on a local EVM; references with unsupported features are simplified as in Table 1.

SmartSpec **achieves comparable gas efficiency**. The Gas(K) and Diff columns in Table 3 report mean per-call gas (thousands of gas) for synthesized code against references. Across all 27 rows, differences range from −7.7% to +13.6%, with mean ≈ +0.7% (footer aggregate). The largest positive gap is ltcSwapAsset +13.6% (*e.g.,* heavy `swapOwnerTx` relative to the reference path); other large positives include auction +9.7% and wallet +10.9%, often coinciding with relational tuple encoding. Several contracts fall below the reference (*e.g.,* erc20 −7.7%, uni −7.2%, voting −6.4%, linktoken −6.8%) despite added instrumentation.

Gas overhead and optimizations. Overhead mainly comes from auxiliary variables for relational state. For example, SmartSpec encodes each relation as a `Tuple`; even `balanceOf` maps an address to a value plus a DSC validity bit that marks records invalidated by newer derivations. Optimizing DeCon's compiler to remove runtime verification variables could reduce this cost. Conversely, some contracts save gas because SmartSpec consolidates modules (*e.g.,* Escrow and Crowdsale in crowdfunding) and orders cheap checks (*e.g.,* `amount > 0`) before costly reads (*e.g.,* `balance[sender] >= amount`).

Maximal permissiveness is critical for generalizability. Without maximal permissiveness, SmartSpec often converges to trivially restrictive implementations, *e.g.,* enforcing both $n > 0$ and $n \leq 0$, which reject all transactions and are vacuously safe but useless. Safety properties alone admit many "correct" programs that miss intended functionality; maximal permissiveness biases search toward safe yet usable behavior.

6.3 Qualitative Comparison with Baseline Tools

To the best of our knowledge, no tool synthesizes smart contracts from examples, relational logic, and temporal logic together. We compare with the two closest tools.

SCSYNT. SCSYNT [19] uses parameterized pastTSL obligations: the user specifies how each transaction updates each affected cell. For example, its ERC-20 specification writes three explicit updates for `approved(m,n)`: `approve(m,n)` ⟶ `approved(m,n)` ⟻ `arg@amount`, `transferFrom(m,n)` ⟶ `approved(m,n)` ⟻ `approved(m,n)-arg@amount`, and `otherwise` ⟶ `approved(m,n)` ⟻ `approved(m,n)`. SmartSpec instead names aggregates once: Figure 5a line 4 defines `balanceOf` from `investTotal` and `refundTotal` (lines 2–3), from which the state update is derived *automatically*. Thus SCSYNT enumerates event-by-event assignments, while SmartSpec states the cell's meaning as investment minus refund. For interpretability, SCSYNT's machines expose numeric states (*e.g.,* `s1`, `s2`); SmartSpec invents no output states, and verification-only variables such as `once[tx]` flags are not emitted.

Partial-program tools. Sketch [34] represents *partial-program* synthesis: it fills holes in a user-supplied skeleton. In contrast, SmartSpec generates the program structure from relational specifications. Sketch specializes in constants inside arithmetic expressions, while SmartSpec synthesizes first-order relational guard formulas. Recent partial-program tools still assume a skeleton, which does not hold here. Learning-based approaches [27] can synthesize sketches, but require substantial computation and have limited accuracy.

6.4 Coverage of Supported Fragment

To assess scope beyond our benchmarks, we evaluate the supported Solidity fragment (Table 1) against the top 50 Ethereum contracts by transaction count, using Sourcify.dev source code [18].

Overall, 22 of 50 contracts (44%) fall within our fragment as-is; an `assume`-block extension for cross-contract interactions raises coverage to 38 (76%). Concretely, for a contract that calls an external function *Ext*, we add:

```
assume(Ext.method(...) == expected_return_value)
```

This assume block axiomatizes the callee's response, so SmartSpec synthesizes only the caller, following standard modular verification practice [28], without changing the core synthesis algorithm. We validated this lightweight extension on 3 contracts: `ERC721` (4 lines, 40 s), `ERC677` (3 lines, 15 s), and `upgradeAgentMigration` (7 lines, 8 s). See full version [9] for detailed analysis.

Automatically deriving external-function abstractions and supporting the remaining 12 contracts, which involve contract creation, recursion, or cryptographic primitives (*e.g.,* Merkle trees), remain interesting future work directions.

Summary. SmartSpec is effective and efficient: 23 of 27 contracts reach equivalence without interactive annotation (85%); the remaining 4 converge within four feedback rounds. Output gas is close to references (mean +0.7%), and SmartSpec improves over the closest tools in automation and interpretability. Concrete-trace encoding reduces synthesis time by >2.5×. The supported fragment covers 44% of the top-50 Ethereum contracts as-is and 76% with lightweight extensions.

7 Related Work

Declarative contracts. DeCon [8] introduces a declarative approach to smart contract implementation, modeling contracts as Datalog database-manipulation programs and compiling them deterministically to Solidity. We extend this idea to deductive sketch generation. Unlike DeCon, which ensures Datalog-to-Solidity semantic equivalence, SmartSpec verifies user-specified safety properties over infinite transaction sequences [7, 28] and integrates search with verification to synthesize verified implementations.

Verification. An efficient verifier is central to SmartSpec's inductive synthesis. Verx [28] verifies temporal-logic safety properties, and DCV [7] verifies declarative smart contracts. SmartSpec uses a Verx-like specification interface and DCV's backend.

When a candidate violates a property, SmartSpec generates counterexamples. EthBMC [20] and ESBMC [35] instead target specific vulnerabilities, while SmartSpec handles general user-specified safety properties.

Program synthesis. Program synthesis techniques [22] have been extensively studied. (1) *Constraint-based synthesis*: SmartSpec's inductive phase follows constraint-based synthesis [24], encoding correctness and syntactic restrictions in one formula whose satisfying assignments are programs. Such techniques have succeeded in program sketching [34], Datalog synthesis [4], database schema refactoring [38], and network configuration synthesis [17]. (2) *Deductive synthesis*: SuSlik [30], Synquid [29], and FIAT [16] target heap-manipulating or functional programs with ownership or Hoare-logic specifications, but not the transaction-based execution model or past-time temporal safety properties central to smart contracts. *(3) Reactive synthesis*: SCSYNT [19] synthesizes smart contracts from TSL specifications but requires one explicit update rule per (transaction, state variable) pair and cannot express aggregated state such as `balanceOf`.

In contrast, SmartSpec exploits three smart-contract properties that make synthesis tractable (Section 3): flat key-value state for relational Datalog, gas costs that bound program complexity, and append-only transaction logs that fit past-time LTL. The approach generalizes to transaction-based systems with flat relational state (*e.g.,* database stored procedures).

8 Conclusion

We present SmartSpec, a multi-modal synthesizer for smart contracts. Developers write relational rules and temporal safety properties; SmartSpec builds a deductive sketch and refines it with counterexample-guided inductive synthesis, biased toward maximal permissiveness. On 27 contracts, runs finish in seconds to minutes; 23 match references without interactive refinement (85%), and the remaining four need at most four feedback rounds. Mean gas overhead is +0.7%, comparable to hand-written references.

Future work includes specifications for contract interactions and synthesis-based repair of vulnerable contracts.

9 Data Availability Statement

All data, source code, benchmarks, and scripts needed to reproduce our results are publicly available in the SmartSpec artifact [9].

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their valuable feedback on this paper. This work was supported by the ShanghaiTech Startup Fund.

References

[1] 2024. Solidity. https://soliditylang.org/.

[2] 2025. Vyper. https://vyperlang.org/.

[3] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap v2 Core. https://app.uniswap.org/whitepaper.pdf.

[4] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-based synthesis of datalog programs. In *Principles and Practice of Constraint Programming: 23rd International Conference*. Springer. doi:10.1007/978-3-319-66158-2_44

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8. doi:10.1109/FMCAD.2013.6679385

[6] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623302

[7] Haoxian Chen, Lan Lu, Brendan Massey, Yuepeng Wang, and Boon Thau Loo. 2024. Verifying Declarative Smart Contracts. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, Lisbon, Portugal, 1–12. doi:10.1145/3597503.3639203

[8] Haoxian Chen, Gerald Whitters, Mohammad Javad Amiri, Yuepeng Wang, and Boon Thau Loo. 2022. Declarative smart contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 281–293. doi:10.1145/3540250.3549121

[9] Tanglin Chen, Haoxian Chen, Huilin Xiang, and Yuepeng Wang. 2026. SmartSpec Artifact: Smart Contract Synthesis via Multi-modal Specifications. Zenodo. doi:10.5281/zenodo.21333365

[10] Xiaohong Chen, Daejun Park, and Grigore Roşu. 2018. A language-independent approach to smart contract verification. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8*. Springer, 405–413. doi:10.1007/978-3-030-03427-6_30

[11] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying Invariant Effectiveness for Securing Smart Contracts. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1772–1795. doi:10.1145/3660786

[12] Coinpaper. 2024. *Weekly Losses from Web3 Exploits Exceeded $138 Million*. https://coinpaper.com/2566/weekly-losses-from-web3-exploits-exceeded-138-million

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] DeFiHackLabs. 2024. GPU. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2024-05/GPU_exp.sol.

[15] DeFiHackLabs. 2024. JokInTheBox. https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/src/test/2024-06/JokInTheBox_exp.sol.

[16] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*. doi:10.1145/2676726.2677006

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part II*. Springer, 261–281. doi:10.1007/978-3-319-63390-9_14

[18] Ethereum Foundation. 2024. Sourcify: Smart Contract Source Verification. https://sourcify.dev.

[19] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. 2023. Reactive synthesis of smart contract control flows. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 248–269. doi:10.1007/978-3-031-45329-8_12

[20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. 2757–2774.

[21] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. doi:10.1145/1925844.1926423

[22] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. doi:10.1561/2500000010

[23] Infosecurity Magazine. 2024. *Cyber-Attacks Drain $1.84bn from Web3 in 2023*. https://www.infosecurity-magazine.com/news/cyber-attacks-drain-web3/

[24] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 215–224. doi:10.1145/1806799.1806833

[25] Moez Krichen, Mariam Lahami, and Qasem Abu Al-Haija. 2022. Formal Methods for the Verification of Smart Contracts: A Review. In *2022 15th International Conference on Security of Information and Networks (SIN)*. IEEE, 01–08. doi:10.1109/SIN56466.2022.9970534

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 254–269. doi:10.1145/2976749.2978309

[27] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *International Conference on Machine Learning*. PMLR, 4861–4870.

[28] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677. doi:10.1109/SP40000.2020.00024

[29] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *PLDI*. doi:10.1145/2908080.2908093

[30] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. In *POPL*. doi:10.1145/3290385

[31] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 107–126. doi:10.1145/2814270.2814310

[32] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-guided synthesis of datalog programs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–27. doi:10.1145/3371130

[33] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 515–527. doi:10.1145/3236024.3236034

[34] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Asian symposium on programming languages and systems*. Springer, 4–13. doi:10.1007/978-3-642-10672-9_3

[35] Kunjian Song, Nedas Matulevicius, Eddie B de Lima Filho, and Lucas C Cordeiro. 2022. Esbmc-solidity: An smt-based model checker for solidity smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 65–69. doi:10.1145/3510454.3516855

[36] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 67–82. doi:10.1145/3243734.3243780

[37] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 452–466. doi:10.1145/3062341.3062365

[38] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/3314221.3314588

[39] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Vol. 12031. Springer, 87–106. doi:10.1007/978-3-030-41600-3_7

[40] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[41] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L Dill. 2020. The move prover. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*. Springer, 137–150. doi:10.1007/978-3-030-53288-8_7

Received 2026-03-26; accepted 2026-06-18