VeriSolid: Correct-by-Design Smart Contracts for Ethereum

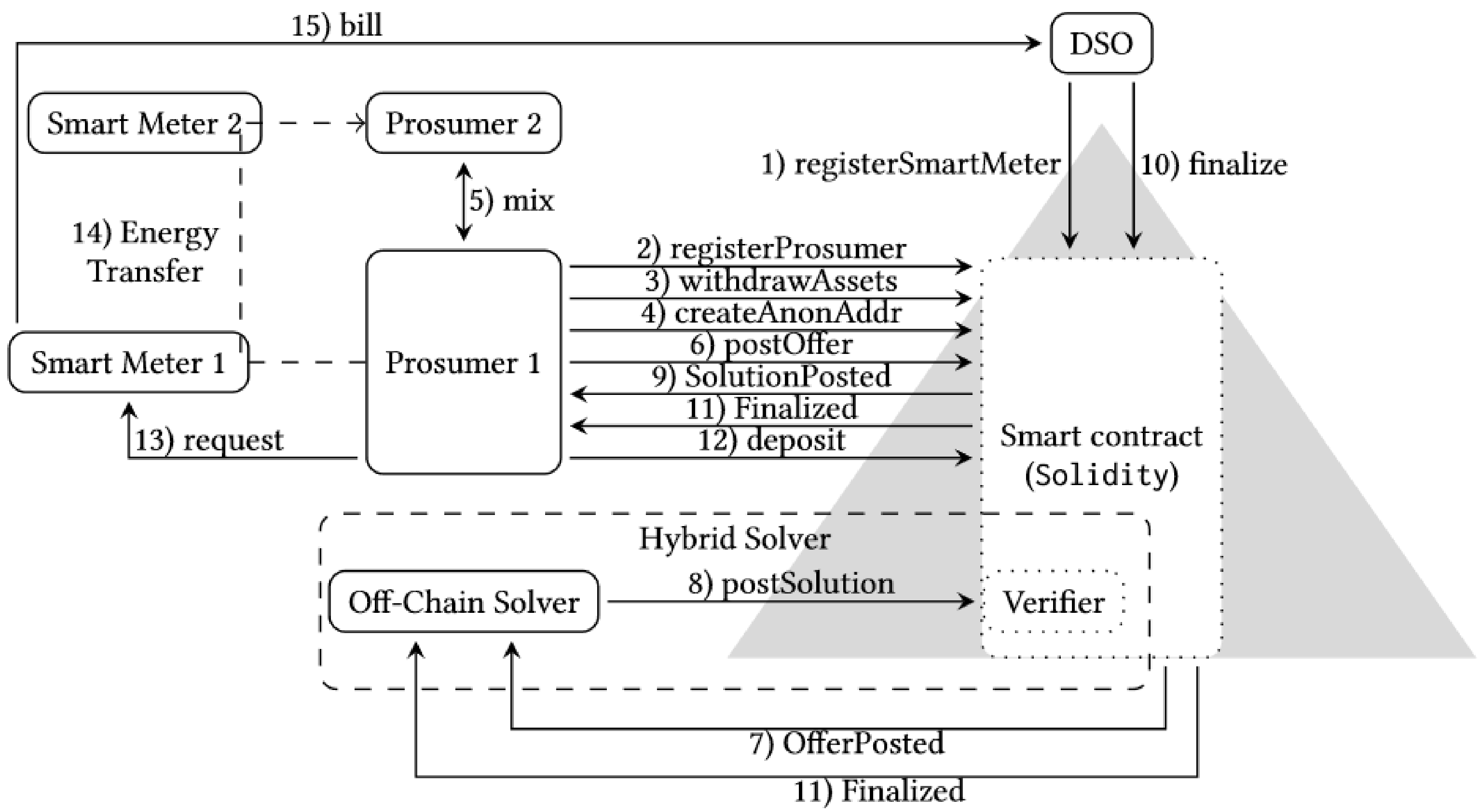
Aron Laszka[1], Anastasia Mavridou[2], Scott Eisele[3], Emmanouela Stachtiari[4], Abhishek Dubey[3]

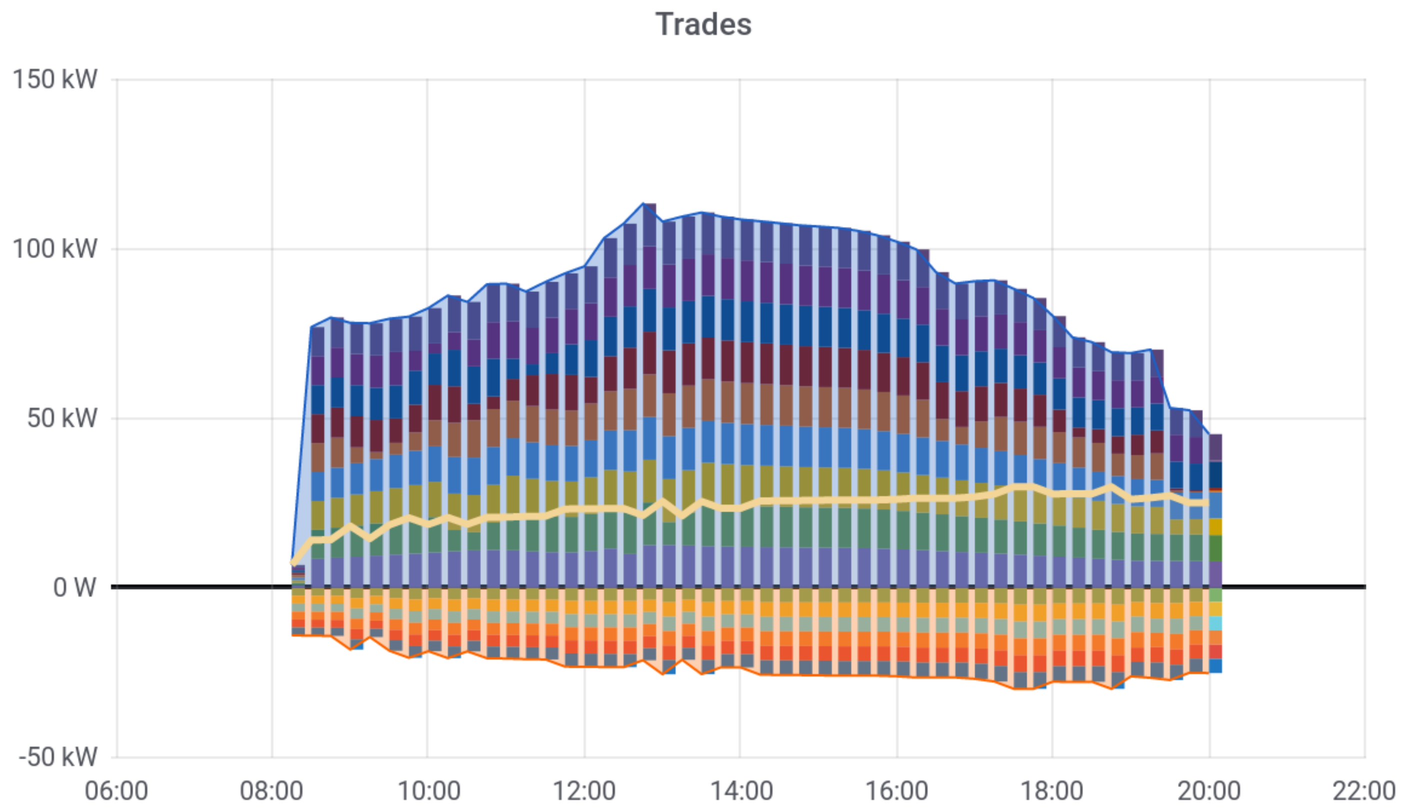
[1] University of Houston
[2] NASA Ames
[3] Vanderbilt University
[4] Aristotle University of Thessaloniki

Smart Contracts on Blockchains

- Smart contract:
  general purpose computation on a blockchain (or other distributed ledger) based computational platform
- Recently popularized by **Ethereum**
  - smart contracts may be developed using high-level languages, such as **Solidity**
  - enables the creation of decentralized applications
- Envisioned to have a **wide range of applications**
  - financial (self-enforcing contracts)
  - Internet of Things
  - decentralized organizations
  - …

Transactive Energy Systems





insecurity

Smart Contract Security ~~in Practice~~

- Notable incidents (amounts vary over time with variations in exchange rate)
  - *The DAO attack*: ~$500 million taken
  - *Parity wallet freeze*: ~$70 million frozen
  - *Parity wallet hack*: ~$21 million taken
- Recent analysis: **34,200 contracts** (out of 1M publicly deployed contracts) have **security issues / vulnerabilities**[1]
- Distributed ledgers are immutable by design
  - smart contract vulnerabilities **cannot be patched***
  - erroneous (or malicious) transactions **cannot be reverted***

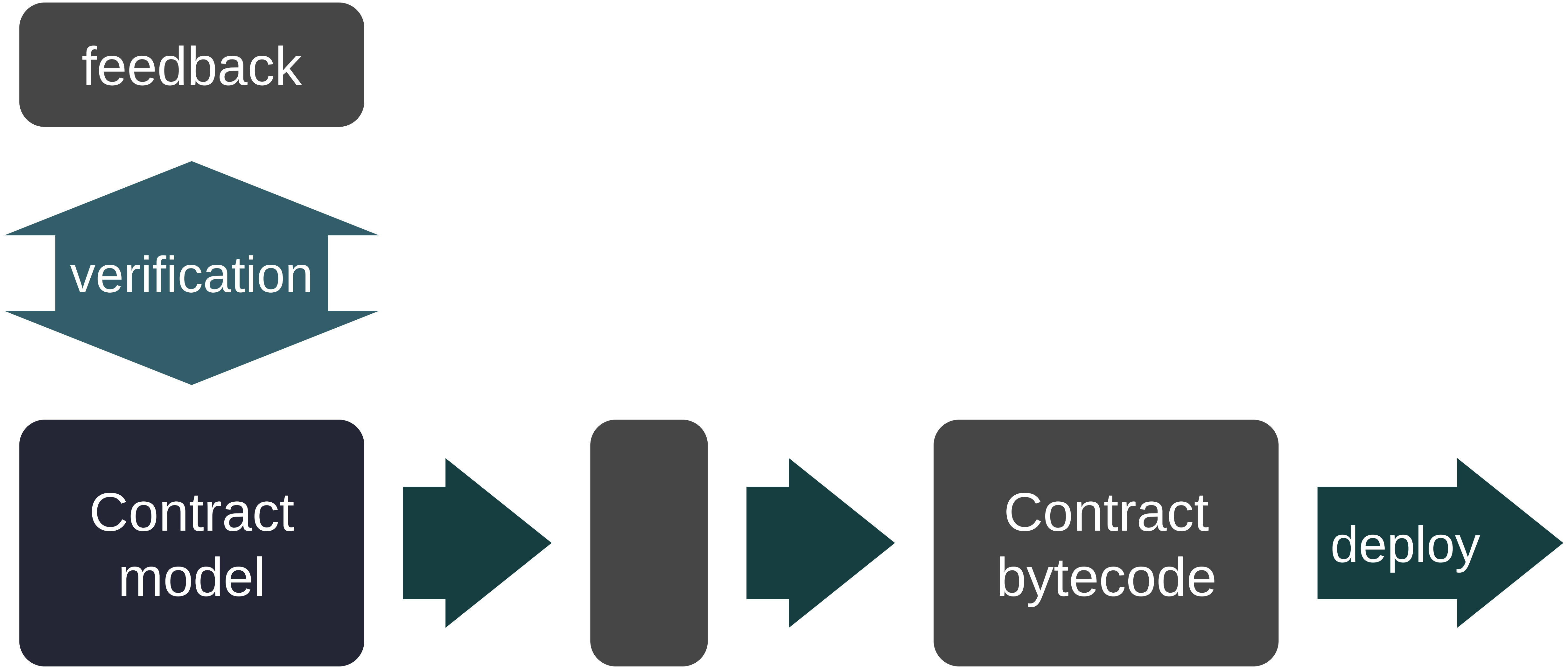
* without undermining the trustworthiness of the contract / ledger

[1] Ivica Nikolic, Aashish KolluriChu, Ilya Sergey, Prateek Saxena, and Aquinas Hobor, "*Finding the greedy, prodigal, and suicidal contracts at scale*," ACSAC'18.

Securing Smart Contracts

- Vulnerabilities often arise due to semantic gap
  - difference between assumptions that developers make about execution semantics and the actual semantics
  - Solidity resembles JavaScript, but it does not work exactly like
- Existing approaches
  - design patterns, e.g., Checks-Effects-Interactions
  - tools for finding (typical) vulnerabilities
    - OYENTE
    - MAIAN
    - …
  - tools for verification and static analysis
    - SECURIFY
    - RATTLE
    - …

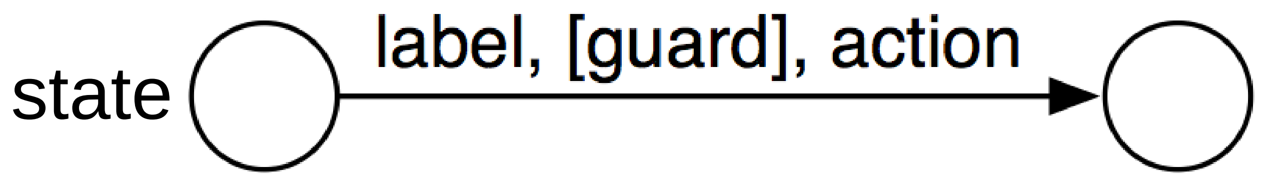
Correct-by-Design Contract Development



- Advantages of model-based approach
  - specification of **desired properties** with respect to a **high-level model**
  - providing **feedback** to developer with respect to a **high-level model**

VERISOLID Model

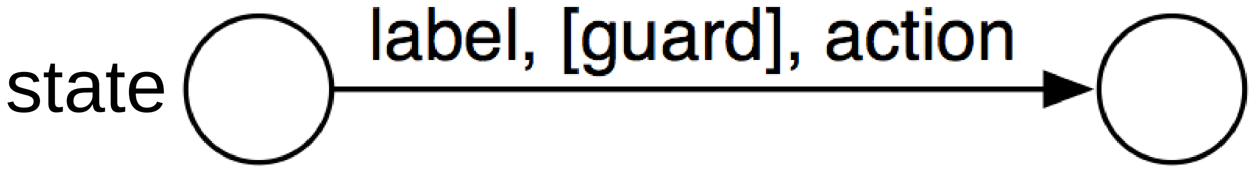
- Formal, transition-system based language for contracts



  - each contract may be represented as a transition system

VeriSolid Model

- Formal, transition-system based language for contracts



  - each contract may be represented as a transition system

Definition: A smart contract is a tuple $(D, S, S_F, s_0, a_0, a_F, V, T)$

- D custom data types and events
- S states
- $S_F \subset S$ final states
- $s_0 \in S$ initial state
- $a_0 \subset \mathbb{S}$ initial action
- $a_F \subset \mathbb{S}$ fallback action
- V contract variables
- T transitions (names, source and destination states, guards, actions, parameter and return types)

$\mathbb{S}$: subset of Solidity statements

implemented as functions in the generated code

VeriSolid Semantics

- We define semantics in the form of Structural Operational Semantics
- Basic transition rule:

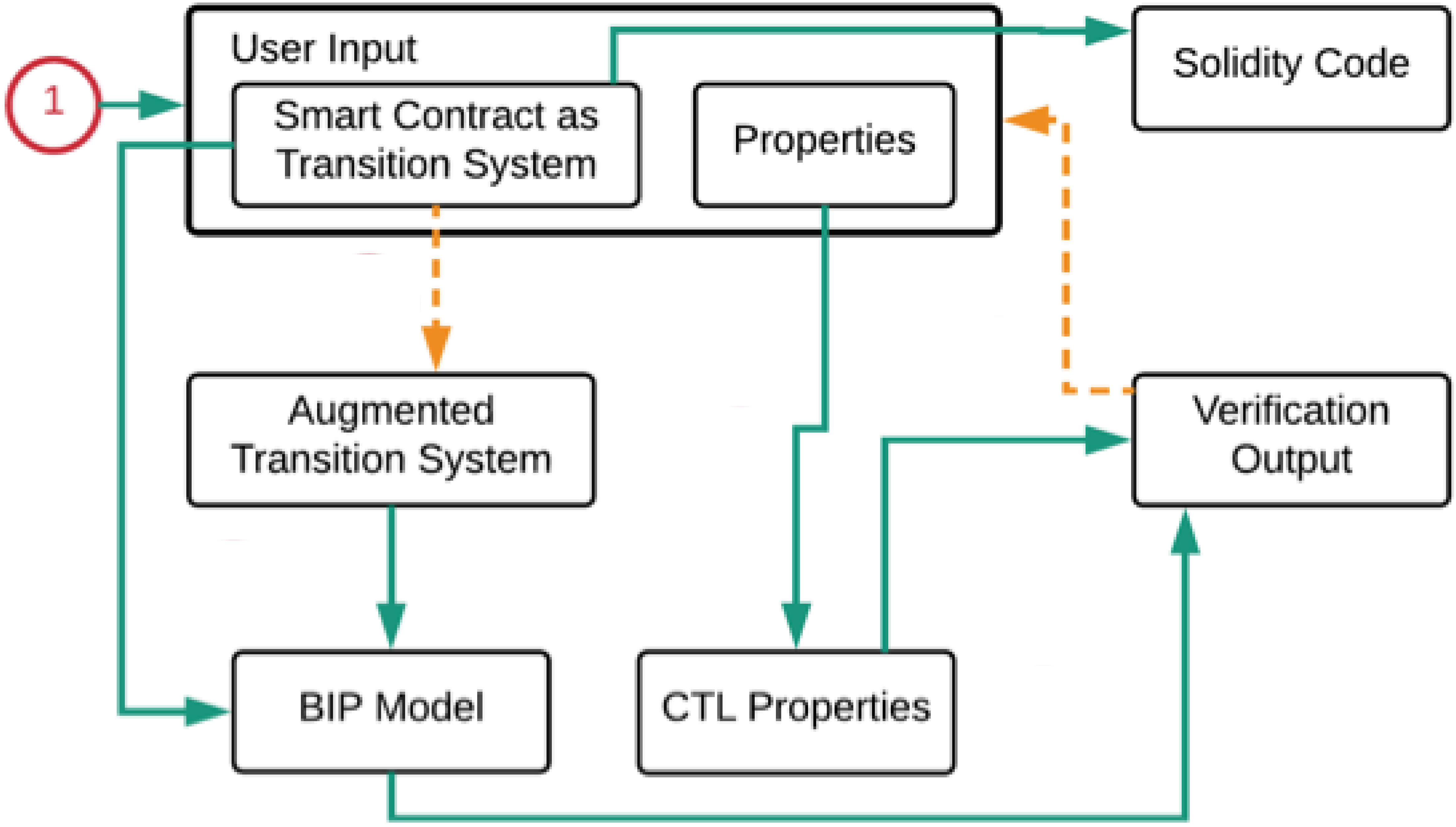
$$\text{TRANSITION}\quad \frac{\begin{array}{c} t \in T, \quad name = t^{name}, \quad s = t^{from} \\ M = Params(t, v_1, v_2, \ldots), \quad \sigma = (\Psi, M) \\ \mathrm{Eval}(\sigma, g_t) \to \langle(\hat{\sigma}, N), \texttt{true}\rangle \\ \langle(\hat{\sigma}, N), a_t\rangle \to \langle(\hat{\sigma}', N), \cdot\rangle \\ \hat{\sigma}' = (\Psi', M'), \quad s' = t^{to} \end{array}}{\langle(\Psi, s), name\,(v_1, v_2, \ldots)\rangle \to \langle(\Psi', s', \cdot)\rangle}$$

  - transition t changes ledger state from Ψ to Ψ' and contract state from s to s'

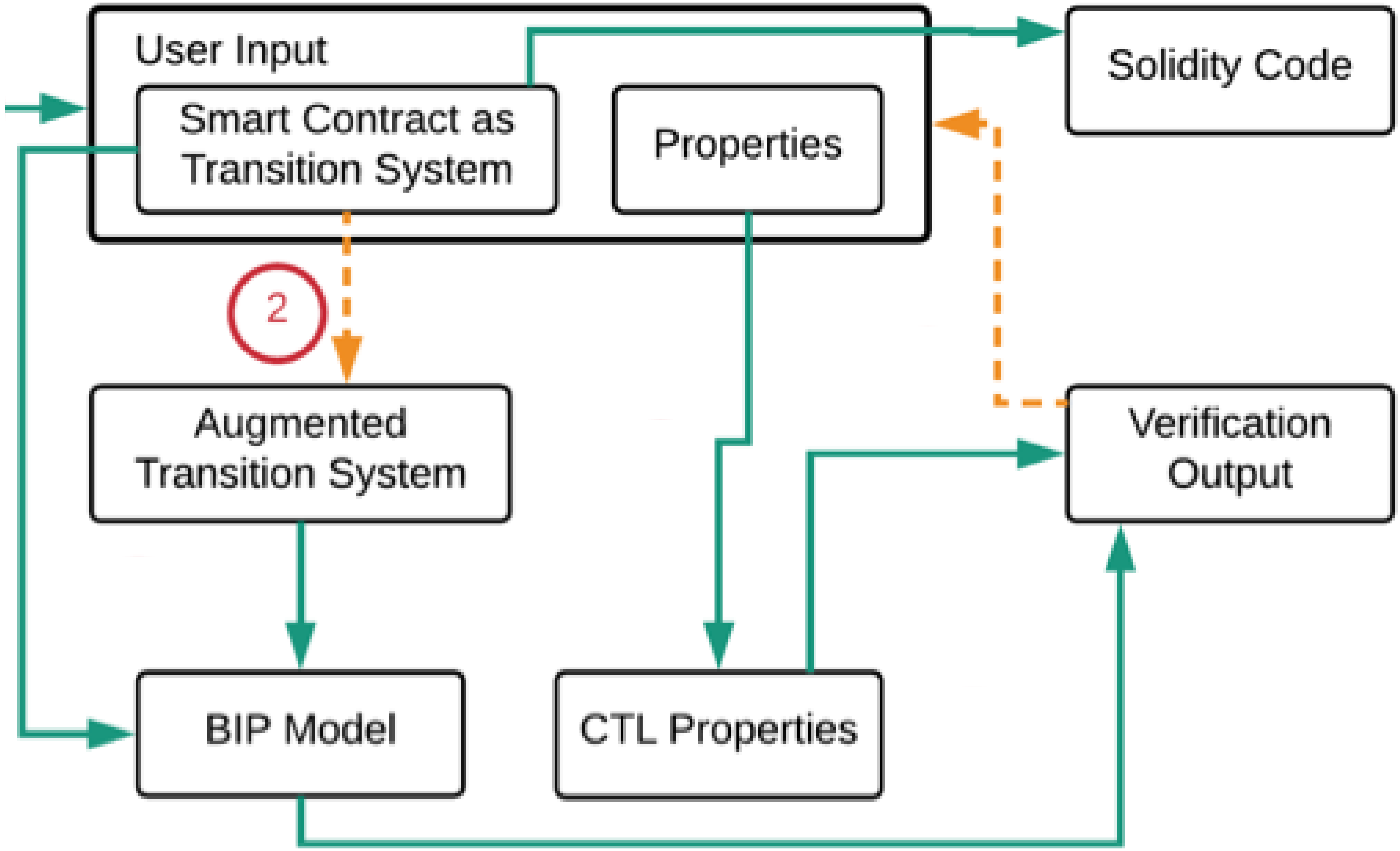
- We also define semantics for erroneous transitions (e.g., exceptions) and for supported Solidity statements $\mathbb{S}$
- Transitions work "*as expected*" from a transition system *

* with Solidity-related additions, such as exceptions and fallback functions

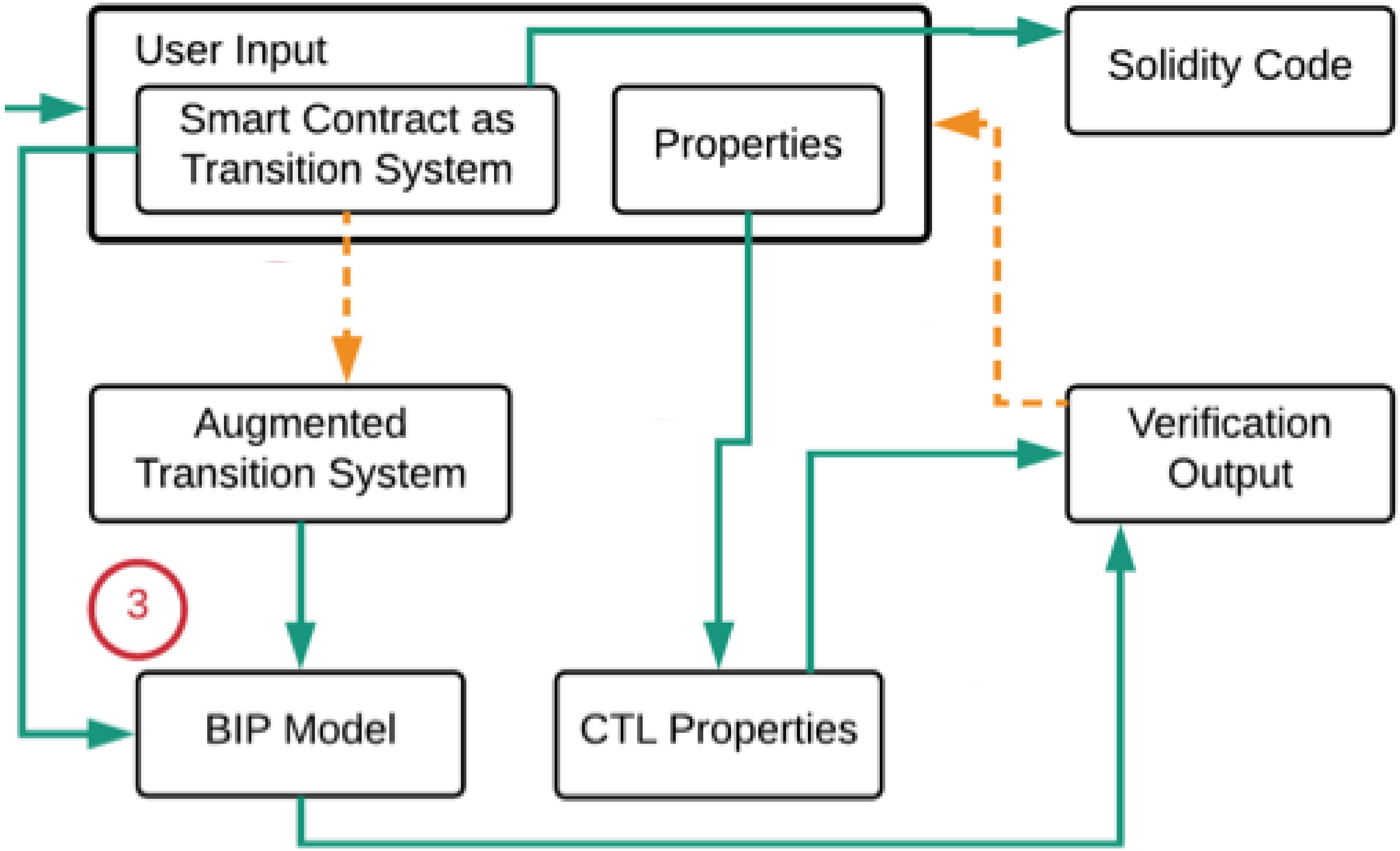
VeriSolid: Correct-by-Design Smart Contracts



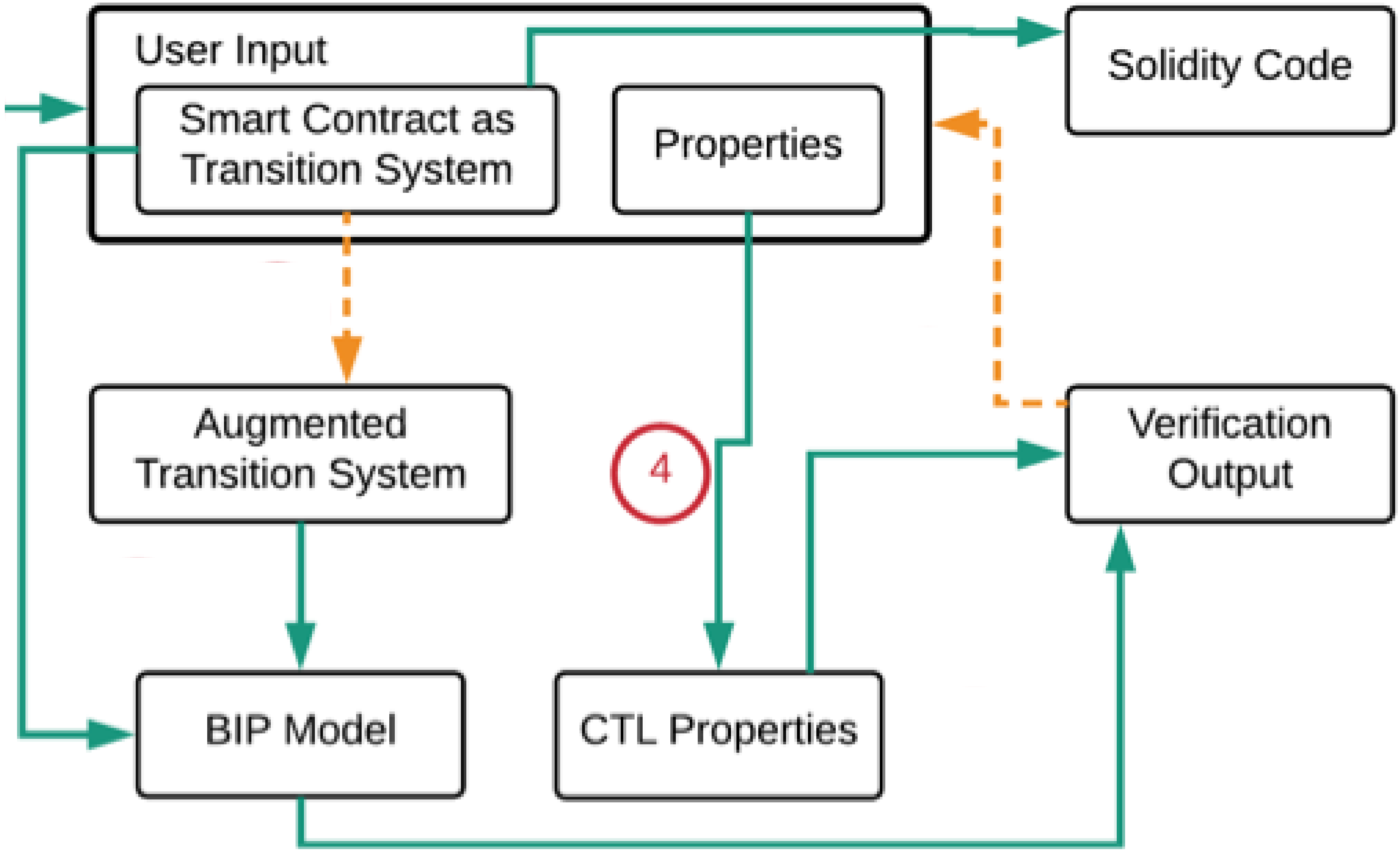
VeriSolid: Correct-by-Design Smart Contracts



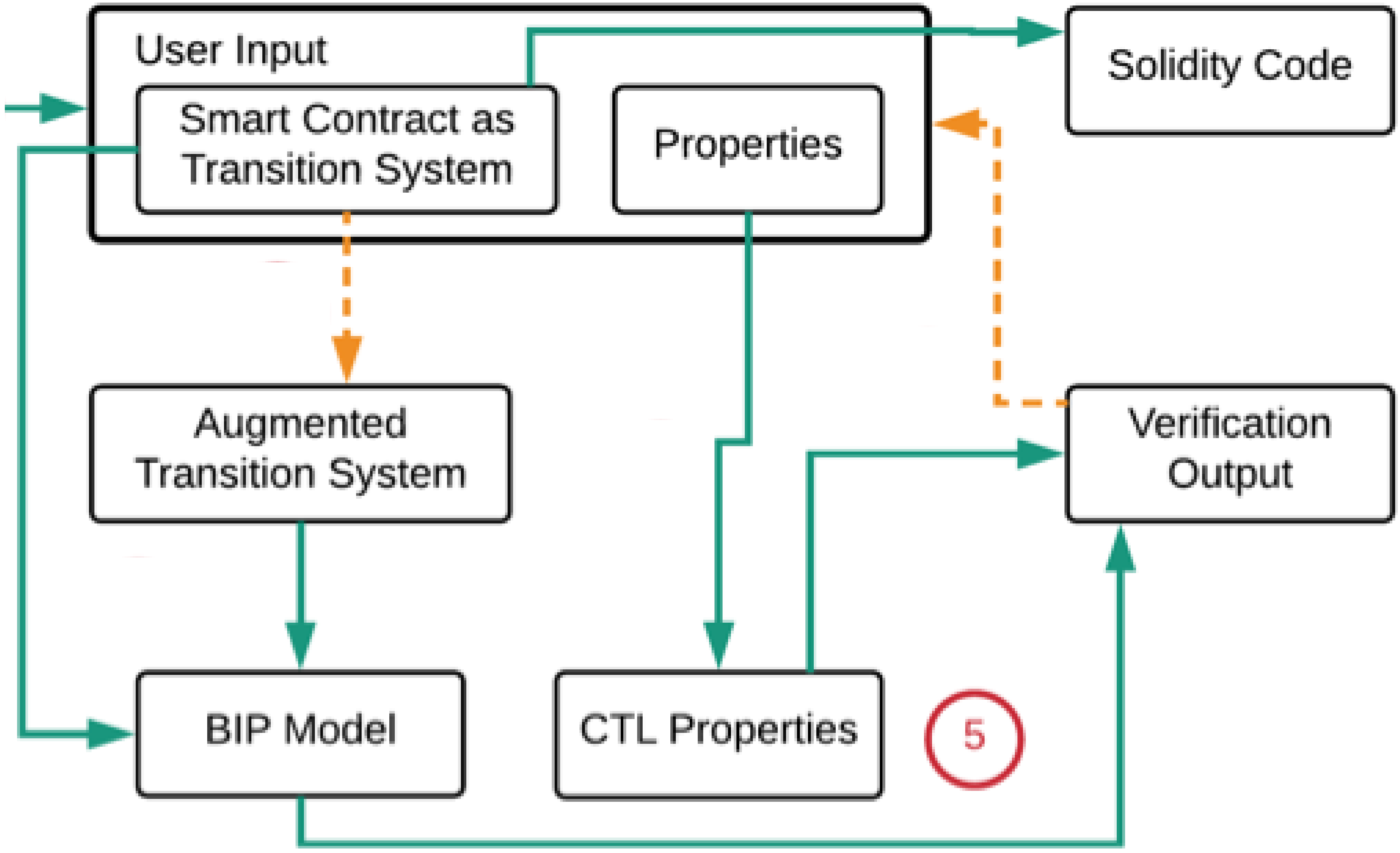
VeriSolid: Correct-by-Design Smart Contracts



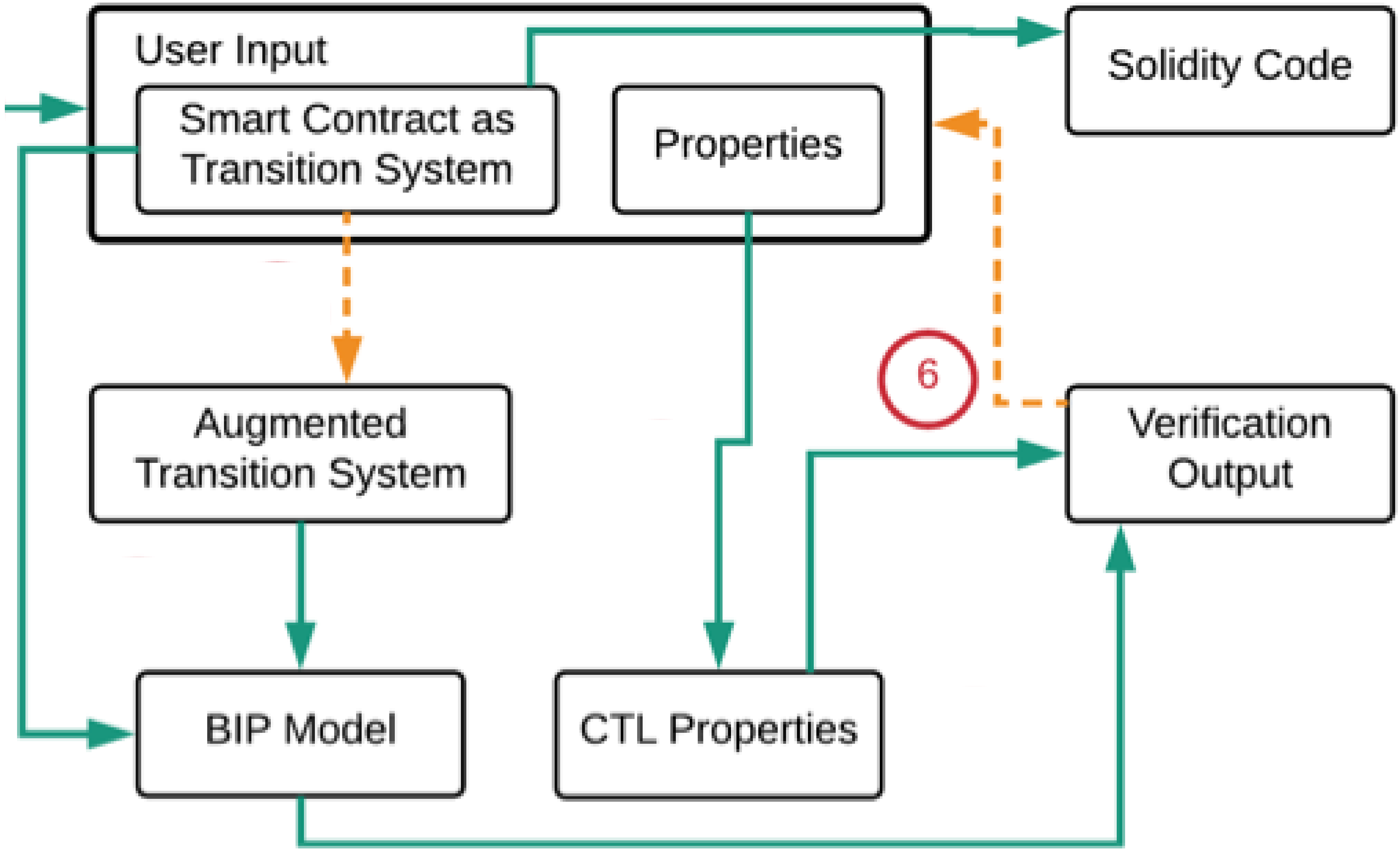
VeriSolid: Correct-by-Design Smart Contracts



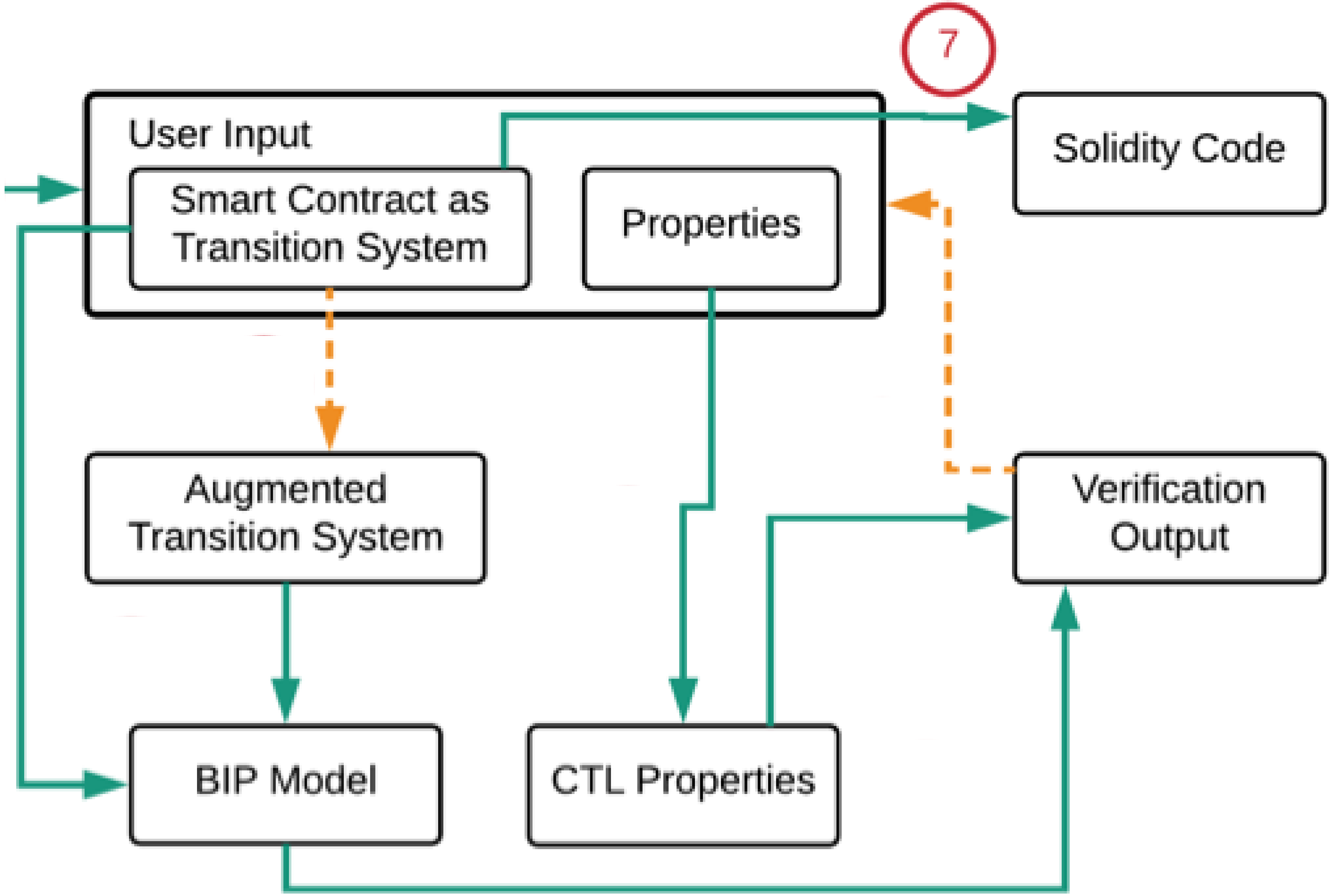
VeriSolid: Correct-by-Design Smart Contracts



VeriSolid: Correct-by-Design Smart Contracts



VeriSolid: Correct-by-Design Smart Contracts



VeriSolid Verification Process

- First, transform a contract into an **augmented transition system**, which captures behavior using transitions
  - based on the formal operational semantics of supported Solidity statements

Theorem: The original contract and the corresponding augmented transition system are observationally equivalent.

- Second, transform an augmented transitions system into an observationally-equivalent **Behavior-Interaction-Priority** (BIP) model
- Over-approximation of contract behavior
  - satisfied safety properties are **satisfied** by the actual contract
  - violated liveness properties are **violated** by the actual contract
- Verification using **nuXmv** model checker

satisfied properties + **violated properties** (with violating transition traces)

VeriSolid Verification

- Instead of searching for vulnerabilities, we verify that a model satisfies **desired properties** that capture **correct behavior**
- **Deadlock freedom**: contract cannot enter a non-final state in which there are no enabled transitions
- Safety and liveness properties
  - specified using Computational Tree Logic (CTL)
  - we provide several CTL templates to facilitate specification

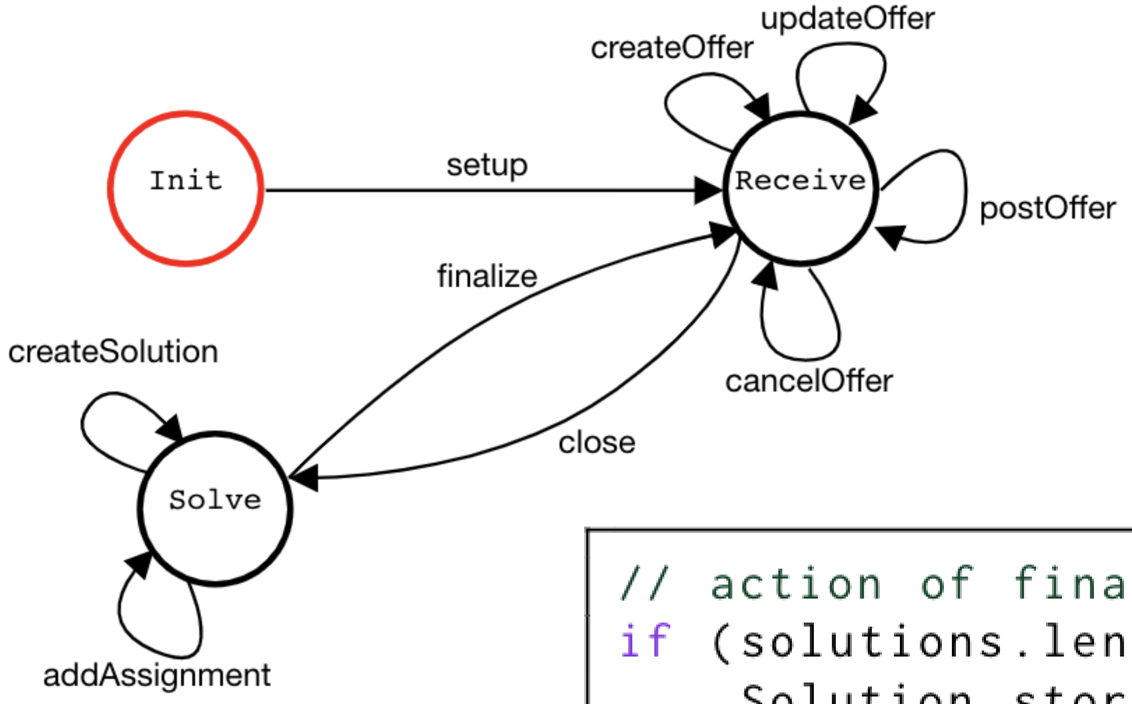
X cannot happen after Y → $AG(Y \rightarrow AG(\neg X))$

where **X** and **Y** can be transitions or statements

  - example:

bid cannot happen after close → $AG(close \rightarrow AG(\neg bid))$

Example Model:
Transactive Energy Market as a Transition System



Violated Property Example

If `close` happens, `postSellingOffer` or `postBuyingOffer` can happen only after `offers.length=0`

```
// action of finalize transition
if (solutions.length > 0) {
    Solution storage solution = solutions[bestSolution];
    for (uint64 i = 0; i < solution.numTrades; i++) {
        Trade memory trade = solution.trades[i];
        emit TradeFinalized(trade.sellingOfferID,
        trade.buyingOfferID, trade.power, trade.price);
    }
    solutions.length = 0;
    offers.length = 0;
}
// offers.length = 0; SHOULD HAVE BEEN HERE
cycle += 1;
```

Conclusion

- VERISOLID advantages
  - high-level model with **formal semantics** (which are familiar to most developers)
  - verification of **desired behavior** (instead of searching for typical vulnerabilities)
  - high-level **feedback** to the developer (for violated properties)
  - Solidity **code generation** (instead of error-prone coding)
- Future work: interactions between multiple contracts

Source code: http://github.com/anmavrid/smart-contracts

Live demo at: http://cps-vo.org/group/SmartContracts
(requires free registration)

Thank you for your attention!

Questions?