Extending SolCMC to Verify Temporal Logic Specifications

From Solidity assertions to temporal specifications for SolCMC model checker

Luigi Bellomarini, Marco Favorito

Applied Research Team @ Bank of Italy
https://bankit.art/

8th DLT Workshop
June 4–6, 2026 — Pula, Italy

Why contract analysis?

high financial stakes
(∼billions of $)

Why contract analysis?

high financial stakes
(∼billions of $)

Why contract analysis?

public code
(...and public bugs)

high financial stakes
(∼billions of $)

Why contract analysis?

public code
(...and public bugs)

permissionless
(also for bad actors)

high financial stakes
(∼billions of $)

Why contract analysis?

public code
(...and public bugs)

permissionless
(also for bad actors)

immutable
(hard to patch)

smart contract security analysis

smart contract security analysis

vulnerability detectors

detect bad patterns

reentrancy | overflows | access control

fast & scalable

limited to known patterns

smart contract security analysis

vulnerability detectors

detect bad patterns

reentrancy · overflows · access control

fast & scalable

limited to known patterns

formal verification

prove correctness against a spec

proof · counterexample

strong guarantees ✨

sometimes automatable (model checking) ✨

often slow

needs specifications

smart contract security analysis

vulnerability detectors

detect bad patterns

reentrancy · overflows · access control

fast & scalable
limited to known patterns

formal verification

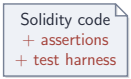
prove correctness against a spec

proof · counterexample

strong guarantees ✨
sometimes automatable (model checking) ✨
often slow
needs specifications

Example: SolCMC (Solidity model checker)

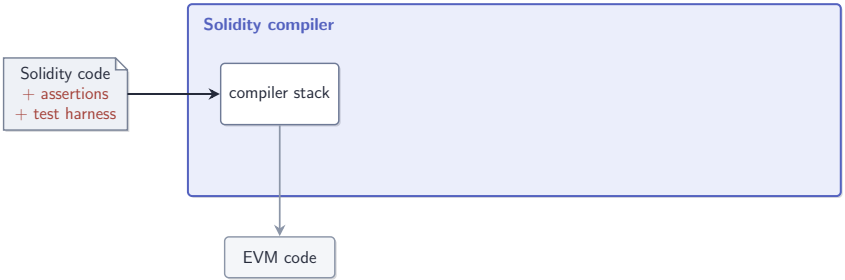
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

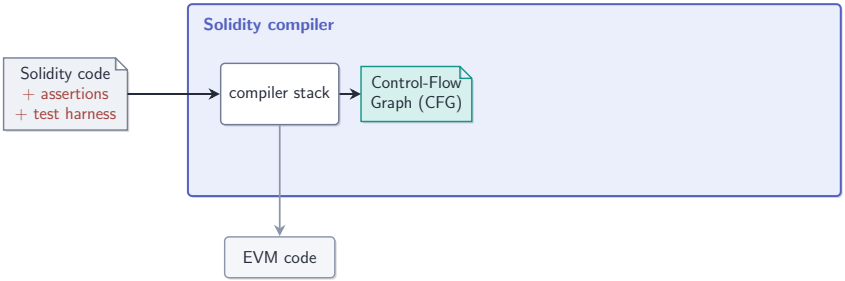
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



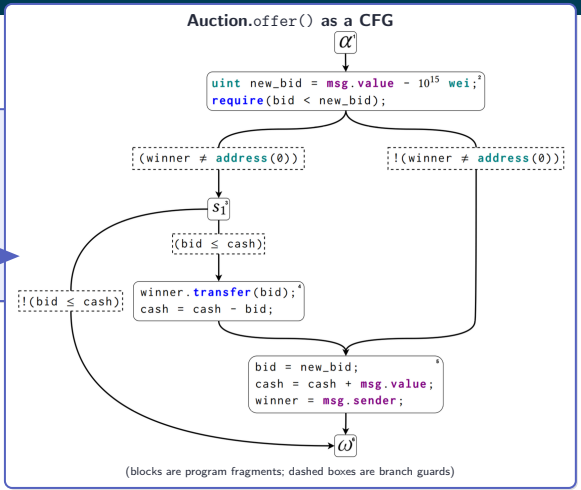
[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]

Auction contract

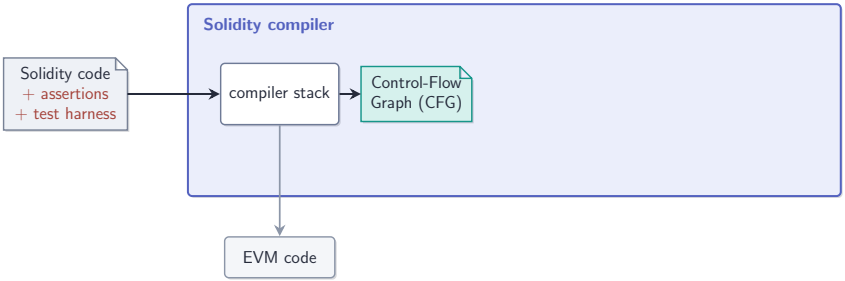
```
1  contract Auction {
2      uint bid = 0;
3      uint cash = 0;
4      address payable winner = address(0);
5
6      function offer() public payable {
7          uint new_bid = msg.value − 10^15 wei;
8          require(bid < new_bid);
9          if (winner ≠ address(0)) {
10             assert(bid ≤ cash);
11             winner.transfer(bid);
12             cash = cash − bid;
13         }
14         bid = new_bid;
15         cash = cash + msg.value;
16         winner = msg.sender;
17     }
18 }
```



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

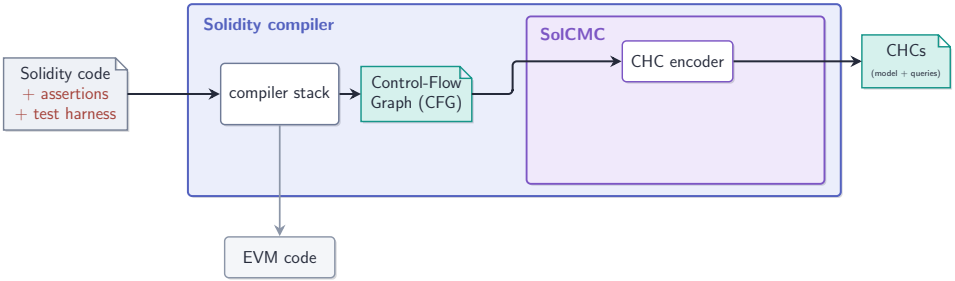
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

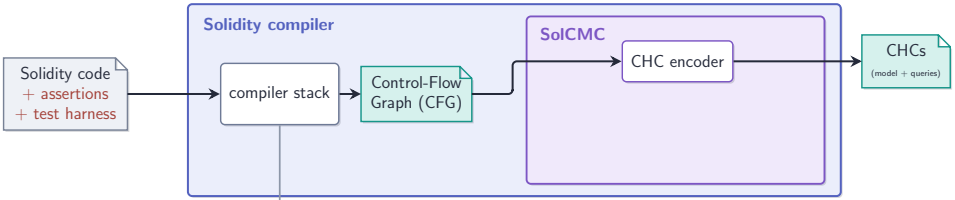
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



Constrained Horn Clauses (CHC)

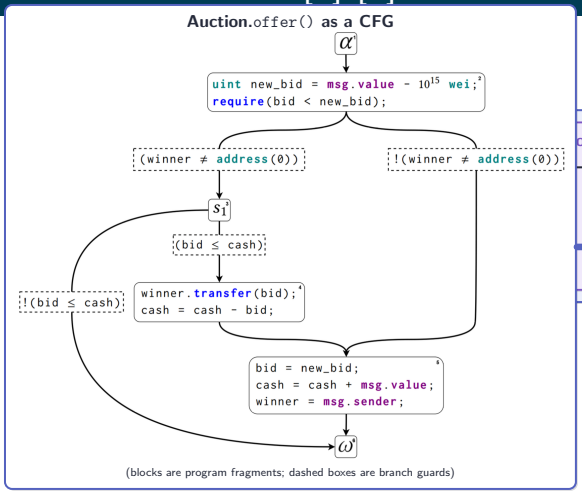
$$\forall \mathcal{V}.\ \left(\underbrace{H(\mathbf{x})}_{\text{head}} \leftarrow \exists \mathbf{y}.\ \underbrace{P_1(\mathbf{y}) \wedge \cdots \wedge P_n(\mathbf{y}) \wedge \phi(\mathbf{x}, \mathbf{y})}_{\text{body}} \right)$$

with $\mathcal{V}$ a set of variables over a background theory $\mathcal{T}$, $\mathbf{x} \cup \mathbf{y} \subseteq \mathcal{V}$, $n \geq 0$

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



Encoding in CHC

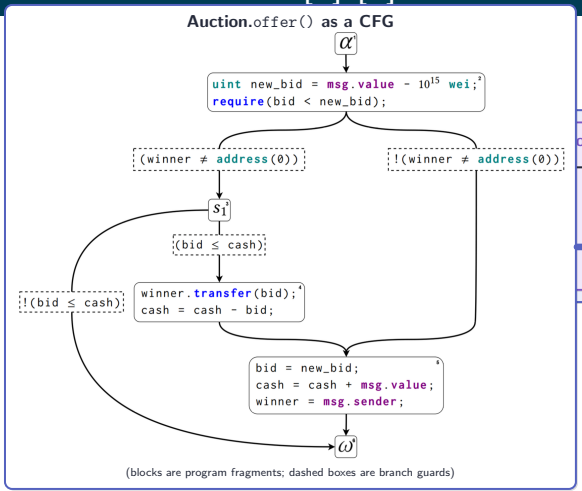
$$C(b,c,w) \leftarrow b = 0 \wedge c = 0 \wedge w = 0$$
$$\mathcal{P}_o^2 \leftarrow \mathcal{P}_o^1$$
$$\mathcal{P}_o^3 \leftarrow \mathcal{P}_o^2 \wedge I_{nb} = I_v - 10^{15} \wedge I_b < I_{nb} \wedge I_w \neq 0$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^2 \wedge I_{nb} = I_v - 10^{15} \wedge I_b < I_{nb} \wedge \neg(I_w \neq 0)$$
$$\mathcal{P}_o^4 \leftarrow \mathcal{P}_o^3 \wedge I_b \leq I_c$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^3 \wedge \neg(I_b \leq I_c) \wedge I_{\tilde{r}} = 3$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^4 \wedge I'_c = I_c - I_b$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^5 \wedge I'_b = I_{nb} \wedge I'_c = I_c + I_v \wedge I'_w = I_s$$
$$\cdots$$
$$C(s') \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} = 0$$
$$\bot \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} \neq 0$$

$$\mathcal{P}_o^n(\underbrace{b,c,w}_{state}, \underbrace{s,v}_{args}, \underbrace{\ell}_{localvars}), \quad n \in \{1..6\}$$

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



Encoding in CHC

$$C(b,c,w) \leftarrow b = 0 \wedge c = 0 \wedge w = 0$$

$$\begin{aligned}
\mathcal{P}^2_o &\leftarrow \mathcal{P}^1_o \\
\mathcal{P}^3_o &\leftarrow \mathcal{P}^2_o \wedge I_{nb} = I_v - 10^{15} \wedge I_b < I_{nb} \wedge I_w \neq 0 \\
\mathcal{P}^5_o &\leftarrow \mathcal{P}^2_o \wedge I_{nb} = I_v - 10^{15} \wedge I_b < I_{nb} \wedge \neg(I_w \neq 0) \\
\mathcal{P}^4_o &\leftarrow \mathcal{P}^3_o \wedge I_b \leq I_c \\
\mathcal{P}^6_o &\leftarrow \mathcal{P}^3_o \wedge \neg(I_b \leq I_c) \wedge I_r = 3 \\
\mathcal{P}^5_o &\leftarrow \mathcal{P}^4_o \wedge I'_c = I_c - I_b \\
\mathcal{P}^6_o &\leftarrow \mathcal{P}^5_o \wedge I'_b = I_{nb} \wedge I'_c = I_c + I_v \wedge I'_w = I_s \\
&\cdots \\
C(s') &\leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} = 0 \\
\bot &\leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} \neq 0
\end{aligned}$$

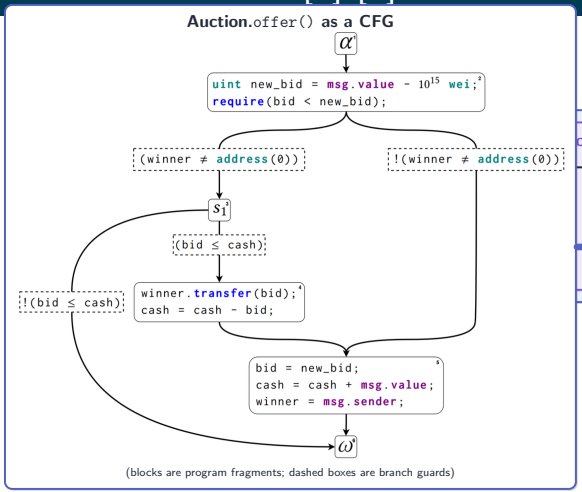
$$\mathcal{P}^n_o(\underbrace{b,c,w}_{state}, \underbrace{s,v}_{args}, \underbrace{\ell}_{localvars}), \quad n \in \{1..6\}$$

Initialization rule:
$C(s) \leftarrow I(s)$

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



Encoding in CHC

$$C(b,c,w) \leftarrow b = 0 \wedge c = 0 \wedge w = 0$$

$$\begin{aligned}
\mathcal{P}^2_o &\leftarrow \mathcal{P}^1_o \\
\mathcal{P}^3_o &\leftarrow \mathcal{P}^2_o \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge l_w \neq 0 \\
\mathcal{P}^5_o &\leftarrow \mathcal{P}^2_o \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge \neg(l_w \neq 0) \\
\mathcal{P}^4_o &\leftarrow \mathcal{P}^3_o \wedge l_b \leq l_c \\
\mathcal{P}^6_o &\leftarrow \mathcal{P}^3_o \wedge \neg(l_b \leq l_c) \wedge l_F = 3 \\
\mathcal{P}^5_o &\leftarrow \mathcal{P}^4_o \wedge l'_c = l_c - l_b \\
\mathcal{P}^6_o &\leftarrow \mathcal{P}^5_o \wedge l'_b = l_{nb} \wedge l'_c = l_c + l_v \wedge l'_w = l_s \\
&\cdots
\end{aligned}$$

$$C(s') \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} = 0$$

$$\bot \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} \neq 0$$

$$\mathcal{P}^n_o(\underbrace{b,c,w}_{\text{state}}, \underbrace{s,v}_{\text{args}}, \underbrace{\ell}_{\text{localvars}}), \quad n \in \{1..6\}$$

Root transaction rule:

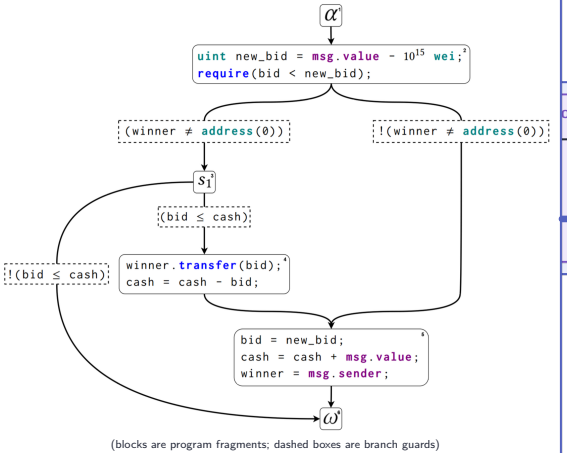
$$C(s') \leftarrow C(s) \wedge S_f(s,a,s',r) \wedge \tilde{r} = 0$$

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]

Auction.offer() as a CFG



(blocks are program fragments; dashed boxes are branch guards)

Encoding in CHC

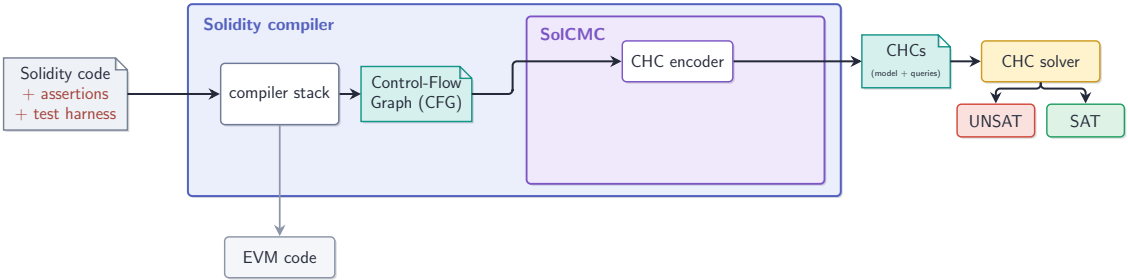
$$C(b,c,w) \leftarrow b = 0 \wedge c = 0 \wedge w = 0$$
$$\mathcal{P}^2_o \leftarrow \mathcal{P}^1_o$$
$$\mathcal{P}^3_o \leftarrow \mathcal{P}^2_o \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge l_w \neq 0$$
$$\mathcal{P}^5_o \leftarrow \mathcal{P}^2_o \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge \neg(l_w \neq 0)$$
$$\mathcal{P}^4_o \leftarrow \mathcal{P}^3_o \wedge l_b \leq l_c$$
$$\mathcal{P}^6_o \leftarrow \mathcal{P}^3_o \wedge \neg(l_b \leq l_c) \wedge l_r = 3$$
$$\mathcal{P}^6_o \leftarrow \mathcal{P}^4_o \wedge l'_c = l_c - l_b$$
$$\mathcal{P}^6_o \leftarrow \mathcal{P}^5_o \wedge l'_b = l_{nb} \wedge l'_c = l_c + l_v \wedge l'_w = l_s$$
$$\cdots$$
$$C(s') \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} = 0$$
$$\bot \leftarrow C(s) \wedge S_o(s,a,s',\tilde{r}) \wedge \tilde{r} \neq 0$$
$$\mathcal{P}^n_o(\underbrace{b,c,w}_{\text{state}}, \underbrace{s,v}_{\text{args}}, \underbrace{\ell}_{\text{localvars}}), \quad n \in \{1..6\}$$

General safety rule:
$$\bot \leftarrow C(s) \wedge S_f(s,a,s',r) \wedge \tilde{r} \neq 0$$

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

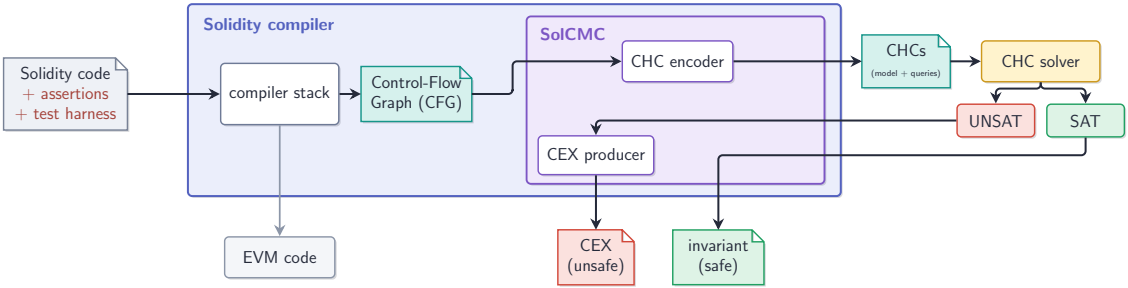
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

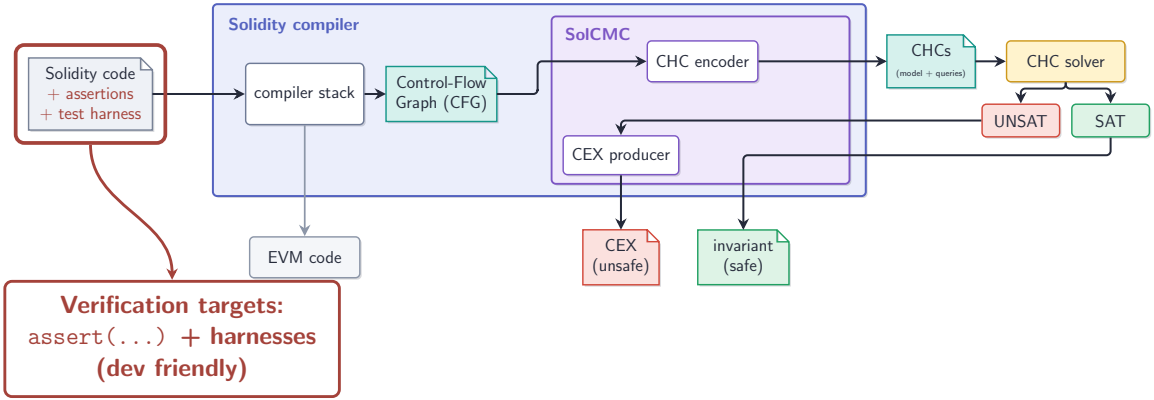
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

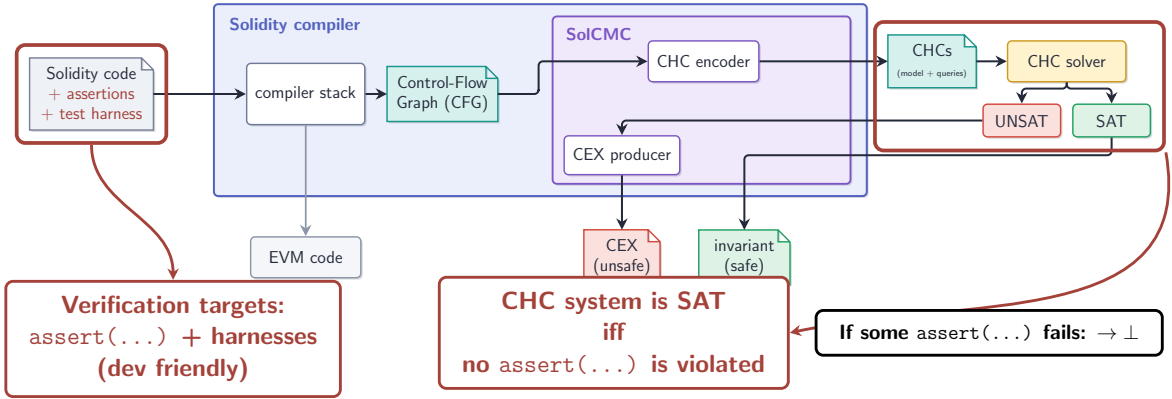
SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC workflow [1] [2]



[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28

SolCMC Limitations

Many requirements constrain the history of calls, but hard to encode them via assertions.

Example: no two (successful) `deposit()` calls in a row [3]

Encode it as a harness:

```
// "no two deposits in a row"
function harness() public {
  deposit();
  deposit();
  assert(false);  // unreachable => property holds
}
```

✗ both are internal self-calls: same `msg.sender`;
→ not a sound encoding! `msg.sender` might differ between calls

Make it sound with ghost variables:

```
function withdraw() public payable {
    // must be updated in every function != deposit()
    g_lastWasDeposit = false;
    ...
}

function deposit() public {
    assert(!g_lastWasDeposit);
    g_lastWasDeposit = true;
    ...
}
```

✓ *can* encode the specification correctly;
✗ intrusive → might change contract behaviour
✗ fragile → hard to mantain when spec/contract changes

[3] M. Bartoletti et al. "Towards Benchmarking of Solidity Verification Tools". In: *FMBC@CAV*. 2024

SolCMC Limitations

Many requirements constrain the history of calls, but hard to encode them via assertions.

Example: no two (successful) `deposit()` calls in a row [3]

Encode it as a harness:

```
// "no two deposits in a row"
function harness() public {
  deposit();
  deposit();
  assert(false);  // unreachable => property holds
}
```

✗ both are internal self-calls: same `msg.sender`;
→ not a sound encoding! `msg.sender` might differ between calls

Make it sound with ghost variables:

```
function withdraw() public payable {
    // must be updated in every function != deposit()
    g_lastWasDeposit = false;
    ...
}

function deposit() public {
    assert(!g_lastWasDeposit);
    g_lastWasDeposit = true;
    ...
}
```

✓ *can* encode the specification correctly;

✗ intrusive → might change contract behaviour

✗ fragile → hard to mantain when spec/contract changes

[3] M. Bartoletti et al. "Towards Benchmarking of Solidity Verification Tools". In: *FMBC@CAV*. 2024

SolCMC Limitations

Many requirements constrain the history of calls, but hard to encode them via assertions.

Example: no two (successful) `deposit()` calls in a row [3]

Encode it as a harness:

```
// "no two deposits in a row"
function harness() public {
  deposit();
  deposit();
  assert(false);  // unreachable => property holds
}
```

✗ both are internal self-calls: same `msg.sender`;
→ not a sound encoding! `msg.sender` might differ between calls

Make it sound with ghost variables:

```
function withdraw() public payable {
    // must be updated in every function != deposit()
    g_lastWasDeposit = false;
    ...
}

function deposit() public {
    assert(!g_lastWasDeposit);
    g_lastWasDeposit = true;
    ...
}
```

✓ *can* encode the specification correctly;
✗ intrusive → might change contract behaviour
✗ fragile → hard to mantain when spec/contract changes

[3] M. Bartoletti et al. "Towards Benchmarking of Solidity Verification Tools". In: *FMBC@CAV*. 2024

Root issue: mixing program with its spec

`assert(...)` is developer-friendly, but not sufficient for complex specifications.

Root issue: mixing program with its spec

`assert(...)` is developer-friendly, but not sufficient for complex specifications.

Goal

How can we equip SolCMC with a separate specification layer...
(with a more expressive specification language)

... while reusing SolCMC's "verification pipeline" as much as possible?

Why extend SolCMC?

- a "high-quality" model of Solidity contracts;
- a strong community behind the Solidity compiler codebase.

Observation: many interesting properties are temporal

Many interesting properties are about sequences of calls

- authorise before settle;
- wait before finalise;
- do not pay the same claim twice;
- close one phase before opening the next.

These are temporal properties, not just local assertions.

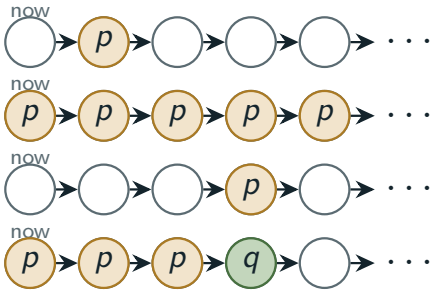
$\longrightarrow$ temporal logic as spec language

What are Temporal logics?

A family of formal languages to reason about properties through time.
Formulas are interpreted over traces (sequences of states)

Examples of temporal operators:

$\mathsf{X}p$	p holds at the **next** step
$\mathsf{G}p$	p must **always** hold
$\mathsf{F}p$	**eventually**, p holds
$p\ \mathsf{U}\ q$	p holds **until** q



Successful specification language for about 50 years since Pnueli's work on LTL [4].

[4] A. Pnueli. "The Temporal Logic of Programs". In: *FOCS*. IEEE Computer Society, 1977, pp. 46–57

The temporal logic LTL_f^{MT}

In this paper, we adopt Linear Temporal Logic on finite traces modulo theories (LTL_f^{MT}) [5] [6].

- interpreted over finite traces $\rightarrow$ semantics is "easier" to handle
- modulo theories $\rightarrow$ faithfully models Solidity data types and constraints

Examples

- **Global invariant:** a recorded balance never exceeds its cap, $\text{G}(\text{balance} \leq \text{cap})$.
- **Call sequencing:** no two successful deposits occur in a row: $\text{G}\,\neg(\text{deposit} \wedge \text{X}\,\text{deposit})$.
- **Conditioned call:** cannot settle before authorisation, $\neg\text{settle}\,\text{W}\,\text{authorise}$.
- **Reachability** there exists an execution where we get an empty balance: $\text{balance} = 0$).

Note: LTL_f^{MT} **model checking** can be reduced to **CHC satisfiability** [6].

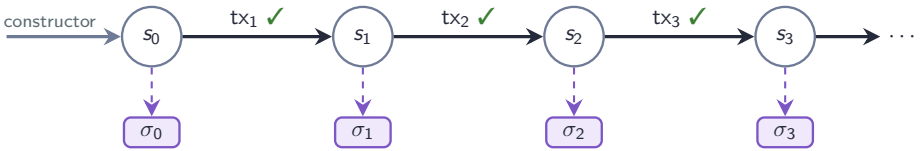
[5] L. Geatti et al. "Linear Temporal Logic Modulo Theories over Finite Traces". In: *IJCAI*. ijcai.org, 2022, pp. 2641–2647

[6] M. Faella and G. Parlato. "A Unified Automata-Theoretic Approach to LTL_f Modulo Theories". In: *ECAI*. 2024

A smart contract as a set of (data) traces

We see a contract C as the set of finite traces of states C can reach

- one step = one successful, top-level public call (a reverts or an internal sub-call add *no* step.)



finite data word $w = \sigma_0\sigma_1\sigma_2\sigma_3\cdots \in \mathrm{Obs}(C)$

- tuple σ_i contains contract state variables + auxiliary variables (e.g., last function called)

Contract satisfaction over traces

A contract C satisfies a specification φ iff all its traces w satisfy φ:

$$C \models \varphi \iff \forall\, w \in \mathrm{Obs}(C).\ w \models \varphi$$

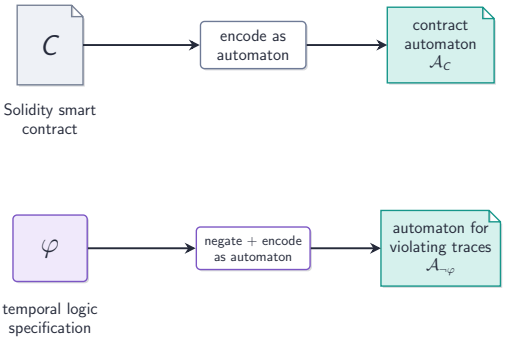
If we followed the classic approach to $\mathrm{LTL}_f^{\mathrm{MT}}$ model checking...

Solidity smart contract

temporal logic specification

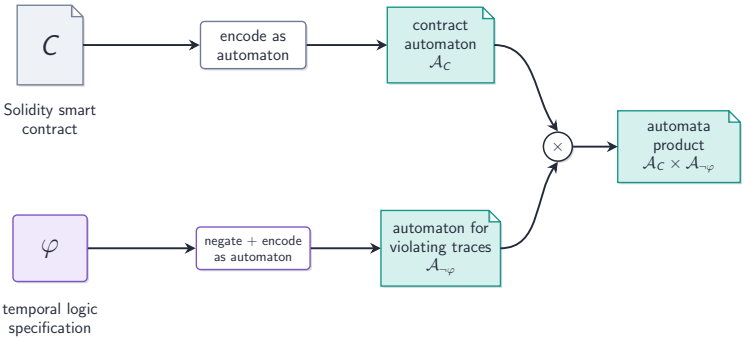
Start from the Solidity contract and the temporal specification.

If we followed the classic approach to $\mathrm{LTL}_f^{\mathrm{MT}}$ model checking...



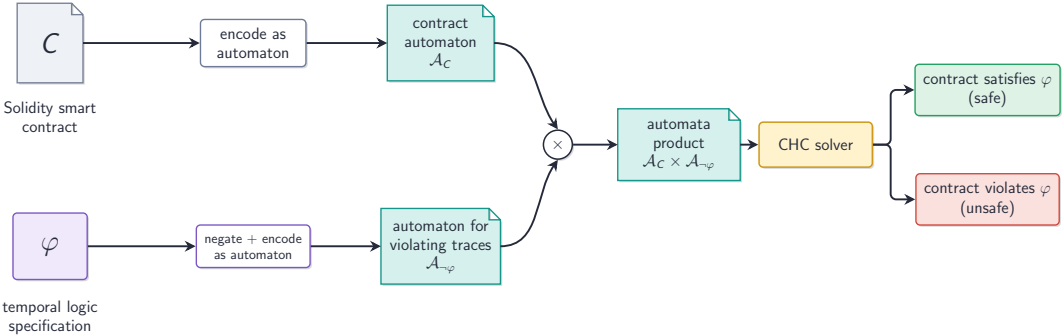
Classic LTL model checking turns both sides into automata.

If we followed the classic approach to LTL_f^{MT} model checking...



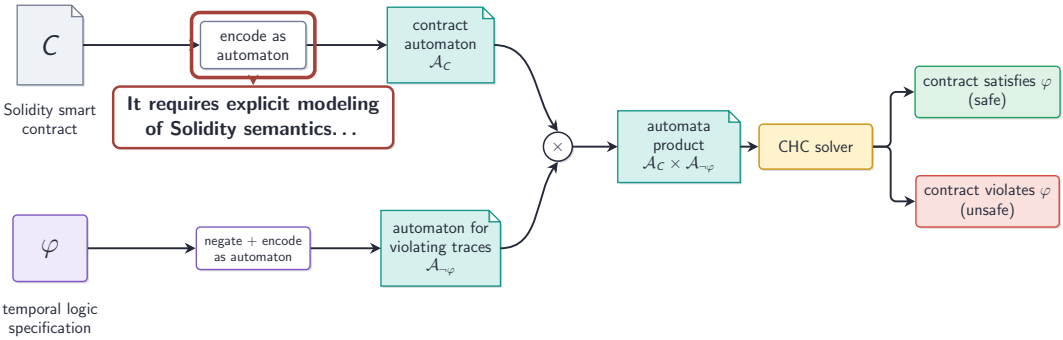
The product automaton $\mathcal{A}_C \times \mathcal{A}_{\neg\varphi}$ accepts all executions of C that satisfy $\neg\varphi$.

If we followed the classic approach to LTL_f^{MT} model checking...



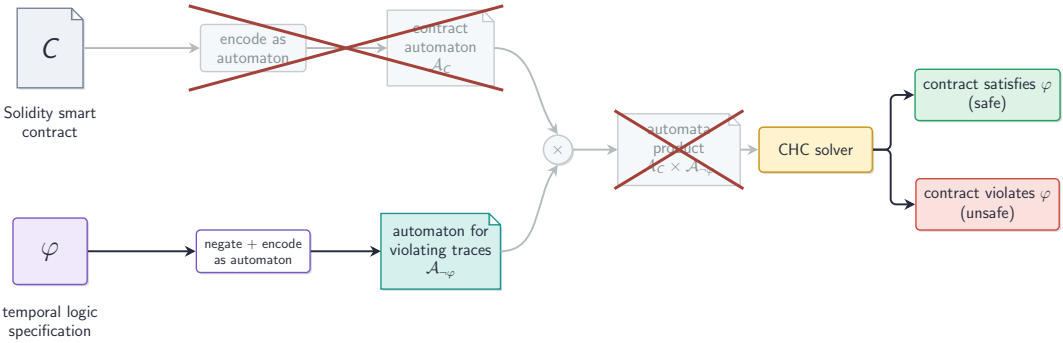
If $\mathcal{L}(\mathcal{A}_C \times \mathcal{A}_{\neg\varphi}) \neq \emptyset \rightarrow$ there exists a counterexample for φ;
Otherwise, φ always holds.

If we followed the classic approach to LTL_f^{MT} model checking...



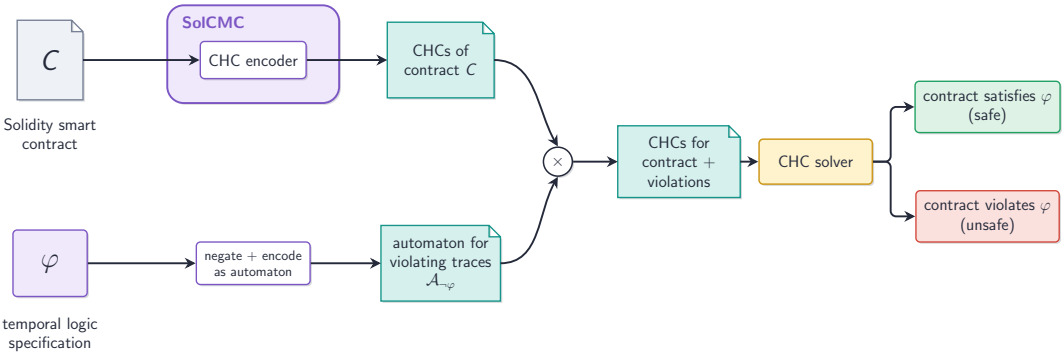
The costly part is the contract side: Solidity semantics must be modeled explicitly.

If we followed the classic approach to LTL_f^{MT} model checking...



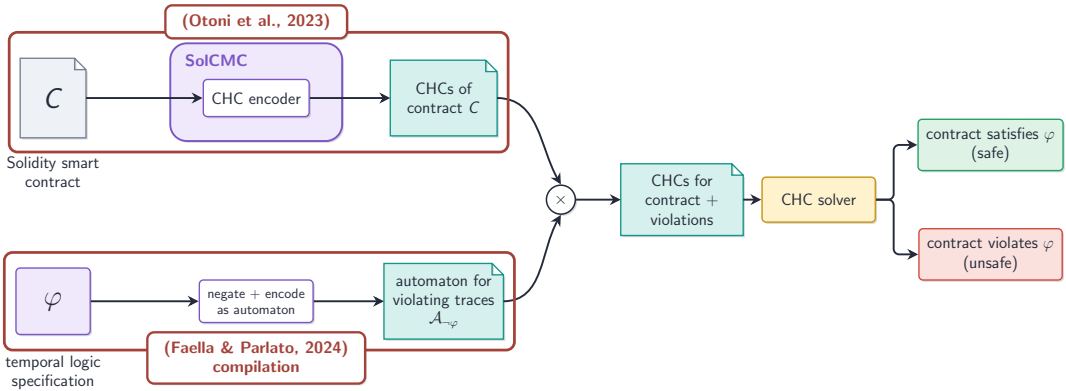
Let's reuse SolCMC's CHC encoding!

Our approach



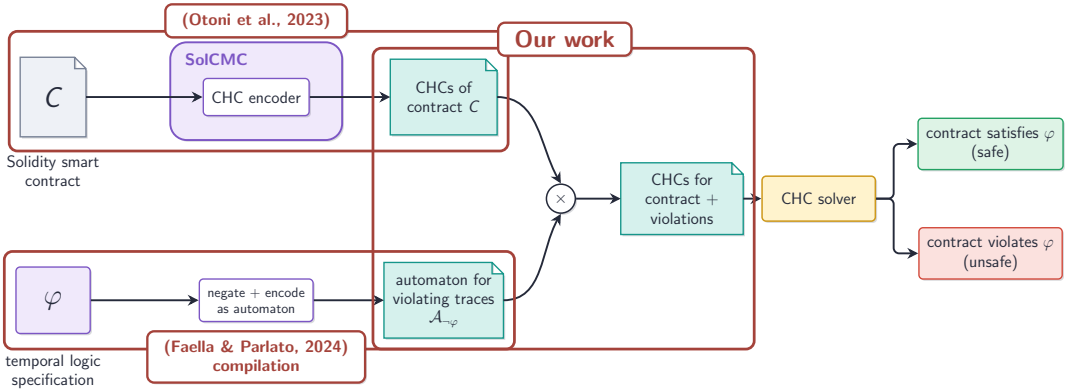
Key idea: cross-product done at the CHC level, reusing SolCMC's contract model

Our approach



Key idea: cross-product done at the CHC level, reusing SolCMC's contract model

Our approach



Key idea: cross-product done at the CHC level, reusing SolCMC's contract model

The CHC product construction

SolCMC CHC encoding Π_C

$$Init_C : C(s) \leftarrow I(s)$$
$$RootTx_{C,f} : C(s') \leftarrow C(s) \wedge S_f(s, a, s', r)$$

Internals of functions
external call rules
SolCMC helper predicates
...
} unchanged

SDWA $\mathcal{A}_{\neg\varphi}$

$$\mathcal{A}_{\neg\varphi} = (\Sigma_C, S^Q, \psi^0, \psi^\Delta, \psi^F)$$

- $\psi^0(q)$ **initial-state formula**
- $\psi^\Delta(q, \sigma, q')$ **transition formula**
- $\psi^F(q)$ **accepting-state formula**

The CHC product construction

SolCMC CHC encoding Π_C

$$Init_C : C(s) \leftarrow I(s)$$
$$RootTx_{C,f} : C(s') \leftarrow C(s) \wedge S_f(s, a, s', r)$$

Internals of functions
external call rules
SolCMC helper predicates
...
} unchanged

SDWA $\mathcal{A}_{\neg\varphi}$

$$\mathcal{A}_{\neg\varphi} = (\Sigma_C, S^Q, \psi^0, \psi^\Delta, \psi^F)$$

- $\psi^0(q)$ **initial-state formula**
- $\psi^\Delta(q, \sigma, q')$ **transition formula**
- $\psi^F(q)$ **accepting-state formula**

$\times$

Product CHC system $\Pi_C \otimes \mathcal{A}_{\neg\varphi}$

$$Init : C^\otimes(s, q) \leftarrow I(s) \wedge ObsInit(s, \sigma) \wedge \psi^0(q_0) \wedge \psi^\Delta(q_0, \sigma, q)$$
$$RootTx_f : C^\otimes(s', q') \leftarrow C^\otimes(s, q) \wedge S_f(s, a, s', r) \wedge Obs_f(s, a, s', r, \sigma) \wedge \psi^\Delta(q, \sigma, q')$$
$$Acc : \bot \leftarrow C^\otimes(s, q) \wedge \psi^F(q)$$

CFG-derived rules are reused unchanged.

The CHC product construction

SolCMC CHC encoding Π_C

$$Init_C : C(s) \leftarrow I(s)$$
$$RootTx_{C,f} : C(s') \leftarrow C(s) \wedge S_f(s, a, s', r)$$

Internals of functions
external call rules
SolCMC helper predicates
...
} unchanged

SDWA $\mathcal{A}_{\neg\varphi}$

$$\mathcal{A}_{\neg\varphi} = (\Sigma_C, S^Q, \psi^0, \psi^\Delta, \psi^F)$$

- $\psi^0(q)$ **initial-state formula**
- $\psi^\Delta(q, \sigma, q')$ **transition formula**
- $\psi^F(q)$ **accepting-state formula**

$\times$

Product CHC system $\Pi_C \otimes \mathcal{A}_{\neg\varphi}$

$$Init : C^\otimes(s, q) \leftarrow I(s) \wedge ObsInit(s, \sigma) \wedge \psi^0(q_0) \wedge \psi^\Delta(q_0, \sigma, q)$$
$$RootTx_f : C^\otimes(s', q') \leftarrow C^\otimes(s, q) \wedge S_f(s, a, s', r) \wedge Obs_f(s, a, s', r, \sigma) \wedge \psi^\Delta(q, \sigma, q')$$
$$Acc : \bot \leftarrow C^\otimes(s, q) \wedge \psi^F(q)$$

CFG-derived rules are reused unchanged.

Key intuition:

- $C(s)$ becomes $C^\otimes(s, q)$: contract state plus automaton state.
- Each committed call computes s' and advances $q \xrightarrow{\sigma} q'$;
- If we visit an accepting q, we get UNSAT $\Rightarrow$ we found a violation of φ

The CHC product construction

SolCMC CHC encoding Π_C

$Init_C : C(s) \leftarrow I(s)$

Product CHC system $\Pi_C \otimes \mathcal{A}_{\neg\varphi}$

Correctness (relative to SolCMC's model)

$$\Pi_C \otimes \mathcal{A}_{\neg\varphi} \text{ is SAT} \iff \forall\, w \in \mathrm{Obs}(C).\ w \models \varphi$$

(the product CHC query is satisfiable *iff* no trace of C accepted by $\mathcal{A}_{\neg\varphi}$ exists)

$\wedge\ \psi^F(q)$

CFG-derived rules are reused unchanged.

$$\mathcal{A}_{\neg\varphi} = (\Sigma_C, S^Q, \psi^0, \psi^\Delta, \psi^F)$$

- $\psi^0(q)$ **initial-state formula**
- $\psi^\Delta(q, \sigma, q')$ **transition formula**
- $\psi^F(q)$ **accepting-state formula**

Key intuition:

- $C(s)$ becomes $C^\otimes(s, q)$: contract state plus automaton state.
- Each committed call computes s' and advances $q \xrightarrow{\sigma} q'$;
- If we visit an accepting q, we get UNSAT $\Rightarrow$ we found a violation of φ

Example: Bank

```
contract Bank {
  mapping(address => uint) balances;

  function deposit() public payable {
    balances[msg.sender] += msg.value;
  }
  function withdraw(uint amount) public {
    require(amount > 0);
    require(amount <= balances[msg.sender]);
    balances[msg.sender] -= amount;
    (bool ok,) = msg.sender.call{value: amount}("");
    require(ok);
  }
}
```

deposit-user-balance

for a fixed sender α: deposit increases its ledger by msgValue

$$\mathsf{G}(\mathsf{X}\,\mathrm{func} = \mathrm{deposit} \wedge \mathsf{X}\,\mathrm{msgSender} = \alpha \rightarrow \mathsf{X}\,\mathrm{balances}[\alpha] = \mathrm{balances}[\alpha] + \mathsf{X}\,\mathrm{msgValue})$$

deposit-contract-balance-lower-bound

contract ETH balance grows by at least msgValue; c = contractAddress

$$\mathsf{G}(\mathsf{X}\,\mathrm{func} = \mathrm{deposit} \rightarrow \mathsf{X}\,\mathrm{ethBalance}[c] \geq \mathrm{ethBalance}[c] + \mathsf{X}\,\mathrm{msgValue})$$

Example: Vault

```
contract Vault {          // two-step withdraw
  enum States { IDLE, REQ }
  address owner; uint wait_time;
  uint request_time; uint amount; address receiver;
  States state;
  function withdraw(address r, uint a) public {
    require(state == States.IDLE);
    require(msg.sender == owner);
    request_time = block.number;
    amount = a; receiver = r; state = States.REQ;
  }
  function finalize() public {
    require(state == States.REQ);
    require(block.number >= request_time + wait_time);
    state = States.IDLE;
    payable(receiver).transfer(amount);
  }
  // + cancel() -> IDLE (recovery key)
}
```

wd-twice

no two successful withdrawals in a row

$$\mathsf{G}\,\neg(\mathsf{func} = \mathsf{withdraw} \wedge \mathsf{X}\,\mathsf{func} = \mathsf{withdraw})$$

relates step i to $i{+}1$ — no single-assert form

finalize-after-wait

finalize only after the recorded waiting period

$$\mathsf{G}(\mathsf{X}\,\mathsf{func} = \mathsf{finalize} \rightarrow \mathsf{X}\,\mathsf{blockNumber} \geq \mathsf{request\_time} + \mathsf{wait\_time})$$

Example: ZeroTokenBet

```
contract ZeroTokenBet {        // a vs b
  address a; address b; address oracle;
  uint timeout_block;
  int balance, balance_a, balance_b;
  // init (balance,balance_a,balance_b) = (1,0,1)
  function deposit() public {          // b only
    require(msg.sender == b);
    require(balance >= 1 && balance_b >= 1);
    balance_b -= 1; balance += 1;
  }
  function win(address dst) public {   // oracle
    require(msg.sender == oracle);
    require(balance >= 2);
    if (dst==a) { balance_a += balance; }
    else        { balance_b += balance; }
    balance = 0;
  }
  function timeout() public {          // split pot
    require(balance >= 2);
    balance_a += 1; balance_b += 1; balance -= 2;
  }
}
```

balance-range

pooled balance remains between 0 and 2

$$G(0 \leq \text{balance} \leq 2)$$

balance-a-range

player A's ledger remains between 0 and 2

$$G(0 \leq \text{balance\_a} \leq 2)$$

balance-b-range

player B's ledger remains between 0 and 2

$$G(0 \leq \text{balance\_b} \leq 2)$$

timeout-balance-zero

some execution reaches a successful timeout with empty pool

$$F(\text{func} = \text{timeout} \wedge \text{balance} = 0)$$

a successful `deposit` by *b*, then `timeout`

Related work (Summary)

- **Certora** is another mature Solidity model checker, but uses procedural specifications and less flexible than LTL_f^{MT} specifications;
- many frameworks support temporal logic too (e.g., **SmartPulse** [7])), but rely on intermediate representation and custom modeling of Solidity semantics (e.g., Boogie/VeriSol)
- other connect smart contracts to mature model checking back ends (**NuSMV/nuXmv** [8, 9], **SPIN** [10], **BIP**, or **CPN tools** [11]).
  - the guarantee is relative to the fidelity of that model/translation to deployed Solidity/EVM behaviour.
- Strategic analyzers: **Solvent** [12]; **KindHML** [13]:
  - Their focus is branching-time properties and strategic reasoning on restricted Solidity fragments;
  - our focus is data-aware linear-time properties over finite execution traces, reusing SolCMC's compiler-generated CHC model.

Conclusion

Take-away

`solcmc-ltl` checks external, data-aware temporal specifications over smart-contract transaction traces by reusing SolCMC's compiler-generated CHC model.

Limitations:

- Ultimately, expressivity is bounded by the underlying SolCMC model: no reverts, no low-level features, untrusted external calls etc.
- no branching-time properties
- CHC solving might be undecidable.

Upside. Because the layer stays close to SolCMC, it can benefit from the Solidity/SolCMC and CHC solving communities.

Future work

- extensive experimental analysis on larger benchmark suites and real contracts;
- attempt-trace semantics for failed calls and revert-aware properties;
- strategic analysis, e.g., two-player games via CHCs;

Extending SolCMC to Verify Temporal Logic Specifications

From Solidity assertions to temporal specifications for SolCMC model checker

Luigi Bellomarini, Marco Favorito

Applied Research Team @ Bank of Italy
https://bankit.art/

8th DLT Workshop
June 4–6, 2026 — Pula, Italy

References I

[1] L. Alt et al. "SolCMC: Solidity Compiler's Model Checker". In: *CAV*. 2022.

[2] R. Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (2023), 15:1–15:28.

[3] M. Bartoletti et al. "Towards Benchmarking of Solidity Verification Tools". In: *FMBC@CAV*. 2024.

[4] A. Pnueli. "The Temporal Logic of Programs". In: *FOCS*. IEEE Computer Society, 1977, pp. 46–57.

[5] L. Geatti, A. Gianola, and N. Gigante. "Linear Temporal Logic Modulo Theories over Finite Traces". In: *IJCAI*. ijcai.org, 2022, pp. 2641–2647.

[6] M. Faella and G. Parlato. "A Unified Automata-Theoretic Approach to LTL_f Modulo Theories". In: *ECAI*. 2024.

[7] J. Stephens et al. "SmartPulse: Automated Checking of Temporal Properties in Smart Contracts". In: *SP*. IEEE, 2021, pp. 555–571.

[8] Z. Nehai, P. Piriou, and F. F. Daumas. "Model-Checking of Smart Contracts". In: *iThings/GreenCom/CPSCom/SmartData*. IEEE, 2018, pp. 980–987.

[9] A. Mavridou et al. "VeriSolid: Correct-by-Design Smart Contracts for Ethereum". In: *Financial Cryptography*. Vol. 11598. Lecture Notes in Computer Science. Springer, 2019, pp. 446–465.

[10] T. Osterland and T. Rose. "Model checking smart contracts for Ethereum". In: *Pervasive Mob. Comput.* 63 (2020), p. 101129.

[11] I. Garfatta et al. "A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts". In: *WETICE*. IEEE, 2021, pp. 69–74.

[12] M. Bartoletti et al. "Solvent: Liquidity Verification of Smart Contracts". In: *IFM*. Vol. 15234. LNCS. Springer, 2024, pp. 256–266.

[13] M. Bartoletti et al. "KindHML: formal verification of smart contracts based on Hennessy-Milner logic". In: *CoRR* abs/2604.14038 (2026).

[14] I. Garfatta et al. "Model checking of vulnerabilities in smart contracts: a solidity-to-CPN approach". In: *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25–29, 2022*. Ed. by J. Hong et al. ACM, 2022, pp. 316–325. DOI: 10.1145/3477314.3507309.

Related work I

- **Certora / CVL** [3]
  - Mature external spec language, supporting invariants and rule programs.
  - Their specifications are procedural, we use declarative specification
  - CVL rules describe fixed scenarios `require; call; assert/satisfy`; $\mathrm{LTL}_f^{\mathrm{MT}}$ formulas more flexible.
  - Certora has richer production features, but has its own Solidity model; we reuse SolCMC CHCs.
  - Certora is closed-source; SolCMC is open-source
- **SmartPulse** [7]
  - Stronger temporal language: SmartLTL supports liveness, fairness, and event predicates for `start`, `finish`, `revert`, and `send`.
  - Also data-aware at the surface: Solidity predicates, old values, and aggregate constructs.
  - SmartPulse instruments the program and propositionalizes temporal atoms; we keep $\mathrm{LTL}_f^{\mathrm{MT}}$ atoms as symbolic theory constraints in the SDWA/CHC product.
  - SmartPulse uses a Boogie/VeriSol + Ultimate Automizer pipeline; we reuse the more lightweight SolCMC's compiler-generated CHC system, and tool-agnostic (Spacer, Eldarica, etc.)
- **External model-checking infrastructures:** NuSMV-based modelling [8], VeriSolid [9], PROMELA/SPIN translations [10], Coloured Petri Nets [11, 14]
  - They connect smart contracts to mature model-checking back ends such as NuSMV/nuXmv, SPIN, BIP, or CPN tools.

Related work II

- They can exploit established temporal/property languages and state-space exploration techniques.
- The verified artifact is usually an external model, a generated transition system, a restricted Solidity fragment, or a Petri-net abstraction.
- Therefore, the guarantee is relative to the fidelity of that model/translation to deployed Solidity/EVM behaviour.
- We avoid introducing another contract semantics: the contract side remains SolCMC's compiler-generated CHC model, and only the temporal monitor is added.

- **Strategic analyzers: KindHML / Solvent** [13] [12]
  - Go beyond linear traces: branching-time logic and liquidity-style strategic reachability.
  - KindHML: first-order Hennessy–Milner logic, via Lustre/Kind 2.
  - Solvent: liquidity, i.e. $\forall$ reachable state, $\exists$ user/transaction sequence achieving an asset-transfer goal.
  - They capture $\forall\exists$-style reasoning that we currently do not support.
  - We instead target linear, data-aware $\mathrm{LTL}_f^{\mathrm{MT}}$ properties over finite successful-call traces, inside SolCMC's CHC pipeline.

Scope and limitations

The method is useful, but the guarantee is intentionally scoped:

1. **Relative to SolCMC's Chc semantics.** Any abstraction or unsoundness (external calls, native ETH, low-level features) is inherited unchanged.
2. **Successful-call finite traces only.** No revert-aware properties, no infinite-run liveness/fairness, no branching-time or strategic notions (liquidity). *Finite-prefix reading*: $\mathsf{G}(p \rightarrow \mathsf{F}q)$ fails on a prefix with p and no later q.
3. **Pointwise observers, incomplete solving.** Mappings / native balance via $m[k]$, balance$[a]$ with bounded lookahead; the CHC solver may return unknown or diverge.

These are not incidental bugs — they define the current semantic contract of the tool.