VC Generation

Overview

Crafted by Arie Gurfinkel

Made by Codex 🤖

github.com/1arie1/ctac

UNIVERSITY OF WATERLOO | FACULTY OF ENGINEERING

Is This Program Safe?

```
entry:
  M := havoc
  i := havoc
  x := M[i]
  ok := x == 0
  assert ok
  halt
```

Can `assert ok` fail?

If so: which values of `M` and `i` break it?

Is This Program Safe?

```
entry:
  M := havoc
  i := havoc
  x := M[i]
  ok := x == 0
  assert ok
  halt
```

Can `assert ok` fail?

If so: which values of `M` and `i` break it?

```
$ ttac vcgen unsafe_bytemap.ttac -solve
sat
```

SAT a failing execution exists.

The Counterexample, Replayed

The solver returns a **model** — concrete values for every symbol:

```
$ ttac vcgen unsafe_bytemap.ttac --solve --model m.txt
sat
i = 1    x = 2    ok = false    M = (lambda addr. 2)
```

The Counterexample, Replayed

The solver returns a **model** — concrete values for every symbol:

```
$ ttac vcgen unsafe_bytemap.ttac –solve –model m.txt
sat
i = 1    x = 2    ok = false    M = (lambda addr. 2)
```

The interpreter replays the model into the failing assert:

```
$ ttac run unsafe_bytemap.ttac –model m.txt
status: assert_failed (assertion failed in block entry)
steps: 5    assert_ok: 0    assert_fail: 1
```

What Did The Solver Actually See?

```
(set-logic QF_UFNIA)

(declare-const i Int)
(declare-const x Int)
(declare-const ok Bool)
(declare-const BLK_EXIT Bool)
(declare-fun M (Int) Int)

; block entry
(assert (= x (M i)))
(assert (= ok (= x 0)))

; cfg constraints
(assert (=> BLK_EXIT (not ok)))
(assert BLK_EXIT)

(check-sat)
```

Not the program — this formula: the **verification condition** (VC).

- assignments became equalities
- the assert became its negation
- control flow became constraints

VC generation is the translation. These lectures are about how it works — and how it goes wrong.

The Contract

VC generation turns a TAC-like program into an SMT query:

$$\text{sat}(\text{VC}(P, A)) \Leftrightarrow \text{there exists an execution reaching failure of } \ A$$

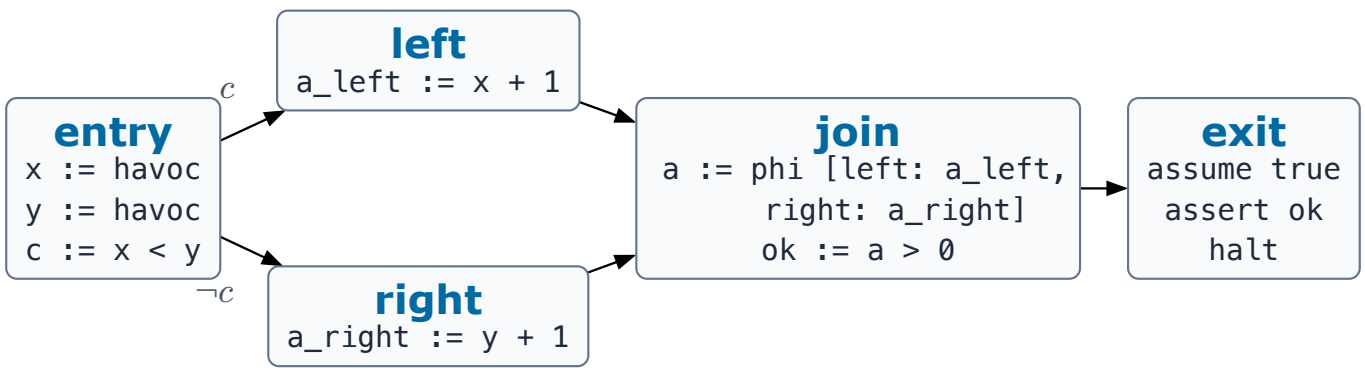
$$\text{SAT} \Rightarrow \text{a bug witness exists} — \text{the model is the failing execution}$$

$$\text{UNSAT} \Rightarrow \text{the assertion is proved in the modeled semantics}$$

Tiny TAC (`ttac`) keeps the semantic core visible: basic blocks and terminators, named branch and assert conditions, havoc, assume, maps, and phi nodes at joins.

The Running Example

One diamond carries the whole lecture series — branches, a join, a phi node, definitions, and a final failure query:



Every encoder in this series processes **this** graph.

Three Encodings of the Same Graph

- **SeaHorn-style** — control flow stays explicit: one reachability variable per block, constrained to describe a path.
- **Boogie-style** — control flow becomes recursion: one backward safety equation per block.
- **SeaBMC-style** — control flow disappears: control-dependence is compiled into data-dependence, then the VC is a flat conjunction.

The encodings differ only in how they represent **path semantics** — definitions, assumes, and the failure query look alike in all three.

Series: 01 Tiny TAC · 02 well-formedness · 03 SeaHorn · 04 Boogie · 05 SeaBMC + comparison · 06 references and borrowing

Tiny TAC

Syntax, Semantics, Demo

Tiny TAC

Tiny TAC (`ttac`) is the small language used in the VCGen notes.

It keeps only the semantic core needed for examples:

- typed registers: `bool`, `int`, `bytemap`
- assignments, havoc, assume, assert
- basic blocks with explicit terminators
- phi nodes at joins

The goal is not to document ctac syntax. The goal is to make the VCGen algorithms precise.

Types And Expressions

Registers have one of three types:

$$\tau ::= \text{bool} \mid \text{int} \mid \text{bytemap}$$

The SMT model is intentionally direct:

$$\text{int} \mapsto \text{Int}$$

$$\text{bytemap} \mapsto \text{Int} \to \text{Int}$$

Integer expressions:

$$\begin{aligned} i ::= {} & n \mid x \mid M[i] \\ & \mid i + i \mid i - i \mid i * i \mid i / i \\ & \mid \text{ite}(b, i, i) \end{aligned}$$

Boolean expressions:

$$\begin{aligned} b ::= {} & \text{true} \mid \text{false} \mid c \\ & \mid i < i \mid i \leq i \mid i = i \mid b = b \\ & \mid \neg b \mid b \wedge b \mid b \vee b \\ & \mid \text{ite}(c, b, b) \end{aligned}$$

`ite` is an expression operator, not a structured statement.

Maps

Loads and stores are expressions over bytemaps.

```
M := havoc
i := havoc
v := havoc
x := M[i]
M2 := M[i := v]
```

Semantically, `M2` agrees with `M` at every address except `i`, where it stores `v`.

Commands

Command forms:

```
x := e
x := havoc
x := phi [B1: x1, B2: x2]
assume b
assert c
```

`:=` is assignment. Equality is a separate expression/operator.

Example:

```
x := havoc
y := havoc
z := havoc
c := x < y
assume y < z
assert c
```

`assume` may use an expression. `assert` uses a named bool register.

Blocks And Control

A complete program — `safe_core.ttac`, the running example of this deck:

```
entry:
  M := havoc
  i := havoc
  limit := havoc
  x := M[i]
  y := x + 1
  M2 := M[i := y]
  c := y <= limit
  if c goto ok else bad

ok:
  assert c
  goto exit

bad:
  assume not c
  goto exit

exit:
  halt
```

Unstructured basic blocks; branches are terminators and branch on a **named** bool register.

Both branches rejoin at `exit` — every block is on an entry-to-exit path. The assert in `ok` only runs on the `c`-true path.

Phi Nodes

Phi nodes select by predecessor block.

```
L:
  xl := 1
  goto J

R:
  xr := 2
  goto J

J:
  x := phi [L: xl, R: xr]
  goto exit
```

- Enter from `L`: `x := xl`
- Enter from `R`: `x := xr`

Demo: First Look At A Program

`ttac stats` reads the surface — commands, types, memory capability:

```
$ ttac stats safe_core.ttac --plain
overview.blocks: 4      overview.commands: 9
command_kinds.Assign: 4   Havoc: 3   Assert: 1   Assume: 1
types.int: 4         types.bool: 1   types.bytemap: 2
memory.capability: bytemap-rw   loads: 1   updates: 1
shape.asserts: 1
```

Four blocks, one assert, read-write bytemap use — matches the program on the previous slide.

Demo: Semantics, Executed

The interpreter **is** the operational semantics. Zero-havoc run:

```
$ ttac run safe_core.ttac --trace
status: done (finished)    assert_ok: 0    assert_fail: 0
entry:
  M := havoc            # havoc bytemap
  i := havoc = 0
  limit := havoc = 0
  x := M[i] = 0         # M[0]
  y := x + 1 = 1
  M2 := M[i := y]       # M[0 := 1]
  c := y <= limit = false
  if c goto ok else bad   # c=false -> bad
bad:
  assume not c          # assume: true
  goto exit             # -> exit
exit:
  halt                  # halt
```

Demo: Semantics, Executed

The interpreter **is** the operational semantics. Zero-havoc run:

```
$ ttac run safe_core.ttac -trace
status: done (finished)    assert_ok: 0    assert_fail: 0
entry:
  M := havoc            # havoc bytemap
  i := havoc = 0
  limit := havoc = 0
  x := M[i] = 0         # M[0]
  y := x + 1 = 1
  M2 := M[i := y]       # M[0 := 1]
  c := y <= limit = false
  if c goto ok else bad   # c=false -> bad
bad:
  assume not c          # assume: true
  goto exit             # -> exit
exit:
  halt                  # halt
```

Havoc defaulted to `0`, so `c` is false and execution takes the `bad` path — the assert never runs. One execution, not all of them.

Demo: Asking About All Executions

`assert c` in block `ok` — can it **ever** fail?

```
$ ttac vcgen safe_core.ttac --solve
unsat
```

UNSAT no execution reaches `ok` with `c` false — the branch guard protects the assert on **every** path.

Demo: Asking About All Executions

`assert c` in block `ok` — can it **ever** fail?

```
$ ttac vcgen safe_core.ttac --solve
unsat
```

UNSAT no execution reaches `ok` with `c` false — the branch guard protects the assert on **every** path.

That is the VCGen contract, seen end to end:

$$\text{VCGen}(P) \text{ is satisfiable} \Leftrightarrow P \text{ has an assertion-failure execution}$$

Interpreter: one chosen execution. Solver: a question over all of them.

VC Generation

Well-Formed Programs

Why Preconditions Exist

The grammar accepts more programs than the encoders should consume.

Well-formedness records the shape expected by each VCGen style:

- CFG shape: entry, exit, paths
- definition discipline: SSA or DSA
- assertion discipline: single final failure query
- join discipline: phi nodes or edge-local dynamic definitions

Each form on the next slides is a **contract with a specific encoder** — watch the "consumed by" notes.

CFG Vocabulary

A CFG has:

- one node per basic block
- edge `b -> b'` when `b` can transfer to `b'`
- successors from outgoing edges
- predecessors from incoming edges

```
entry:
  c := havoc
  if c goto left else right

left:
  goto join

right:
  goto join
```

SESE

A program is **single-entry single-exit** when:

- `entry` and `exit` are distinguished and distinct
- execution starts only at `entry`
- normal completed executions end at `exit`
- every block lies on an entry-to-exit path

$$\text{SESE} = \text{no unreachable blocks, no dead regions, one normal exit}$$

Consumed by: every encoder in this series.

SSA

Static single assignment:

- every register is assigned at most once
- phi assignments are a prefix of the join block
- every phi has one incoming value per predecessor

```
left:
  x_left := 1
  goto join

right:
  x_right := 2
  goto join

join:
  x := phi [left: x_left, right: x_right]
```

Consumed by: SeaHorn-style (deck 03) and SeaBMC-style (deck 05).

DSA

Dynamic single assignment pushes the merge onto incoming edges.

```
left:
  x_left := 1
  x := x_left
  goto join

right:
  x_right := 2
  x := x_right
  goto join

join:
  goto exit
```

The two assignments to `x` are allowed because they occur in sibling predecessors, immediately before the join.

Consumed by: Boogie-style (deck 04).

SSA To DSA

Before:

```
join:
  x := phi [p1: v1, p2: v2]
  goto exit
```

After:

```
p1:
  x := v1
  goto join

p2:
  x := v2
  goto join
```

DSA can be read as placing the merge assignment on the incoming CFG edge. The reverse direction collects sibling dynamic assignments into a phi.

SASA

Single-assume single-assert form puts the query at the exit:

```
exit:
  assume pre
  assert post
  halt
```

- exactly one `assume`, exactly one `assert`, both in `exit`
- `pre` accumulates path assumptions
- `post` summarizes assertion obligations

Consumed by: every encoder — the failure query lives in one place.

Reducing To SASA

Scattered assumes feed `pre`; scattered asserts conjoin into `post`:

Before — two asserts, one assume:

```
entry:
  x := havoc
  assume x > 0
  c1 := x > 0
  assert c1
  y := x + 1
  c2 := y > 1
  assert c2
  halt
```

After — one query at `exit`:

```
entry:
  x := havoc
  c1 := x > 0
  y := x + 1
  c2 := y > 1
  post := c1 and c2
  goto exit

exit:
  assume x > 0
  assert post
  halt
```

If either assert fails, `post` is false — the failure is preserved. Ordering subtleties are why this is **preprocessing**, done once, before any encoder runs.

Safety Query

For a SASA program, safety is:

$$\forall \xi.\ \text{entry-to-exit}(\xi) \Rightarrow (\text{pre}(\xi) \Rightarrow \text{post}(\xi))$$

The VC asks for the negation:

$$\exists \xi.\ \text{entry-to-exit}(\xi) \wedge \text{pre}(\xi) \wedge \neg\ \text{post}(\xi)$$

Which Encoder Wants Which Form

Form	What it fixes	Consumed by
SESE	CFG shape: no dead regions, one normal exit	all three encoders
SSA	one definition per register; merges at phis	SeaHorn (03), SeaBMC (05)
DSA	merges as edge-local sibling assignments	Boogie (04)
SASA	one final `assume pre; assert post` query	all three encoders

The conversions (SSA ↔ DSA, fold-to-SASA) are semantic preprocessing — encoder choice starts **after** them.

VCGen

SeaHorn Style

Idea

SeaHorn-style VCGen keeps control flow explicit.

For each block `i`, introduce a Boolean reachability variable:

$$\mathtt{bb}_i$$

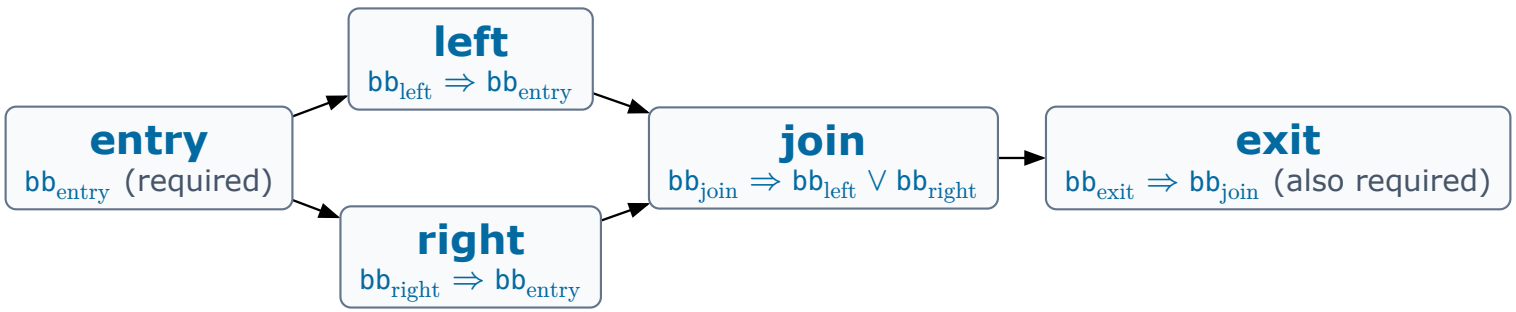
Then constrain the selected blocks so they describe an entry-to-exit path.

$$\mathrm{VCGen\_SeaHorn}(P) = \mathbf{CFG} \wedge \mathbf{DEF} \wedge \mathbf{JUMPS} \wedge \mathbf{PHI} \wedge \mathbf{ERROR}$$

- `CFG`: selected block variables form a path.
- `DEF`: assignments hold when their block is selected.
- `JUMPS`: selected branch edges satisfy branch conditions.
- `PHI`: selected incoming edges choose phi values.
- `ERROR`: the selected exit violates the assertion.

The Diamond, Encoded On The Picture

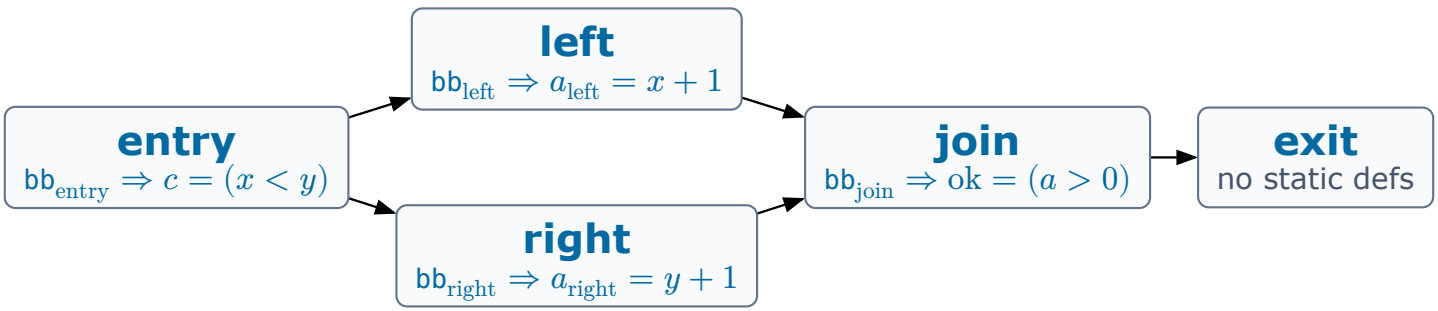
Each component lives somewhere on the graph:



CFG — every selected block has a selected predecessor; `entry` and `exit` are forced.

The Diamond, Encoded On The Picture

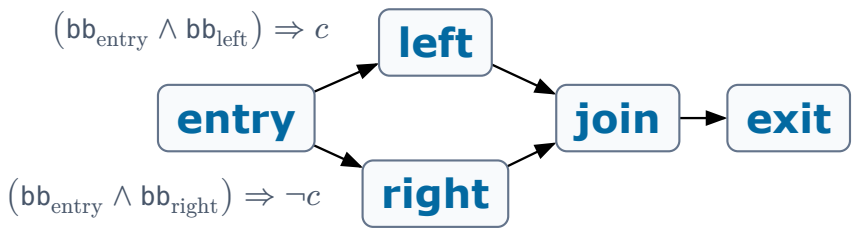
Each component lives somewhere on the graph:



DEF — assignments become equalities, guarded by their block. Havoc contributes nothing.

The Diamond, Encoded On The Picture

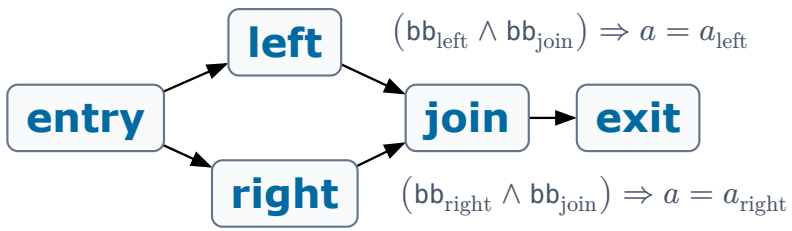
Each component lives somewhere on the graph:



JUMPS — branch conditions attach to the **edges** of the conditional terminator.

The Diamond, Encoded On The Picture

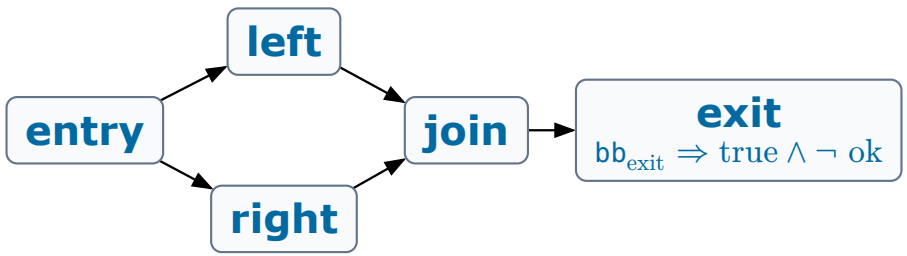
Each component lives somewhere on the graph:



PHI — the selected incoming edge chooses the phi value. (The DSA idea, expressed on edges.)

The Diamond, Encoded On The Picture

Each component lives somewhere on the graph:



ERROR — the selected exit must satisfy `pre` and falsify `post`. With $\mathtt{bb}_{\mathrm{exit}}$ forced, a model is a complete failing execution.

The Whole Formula

$$
\begin{aligned}
\textbf{CFG} &= \mathtt{bb}_{\text{entry}} \wedge \left(\mathtt{bb}_{\text{left}} \Rightarrow \mathtt{bb}_{\text{entry}}\right) \wedge \left(\mathtt{bb}_{\text{right}} \Rightarrow \mathtt{bb}_{\text{entry}}\right) \\
&\quad \wedge \left(\mathtt{bb}_{\text{join}} \Rightarrow \mathtt{bb}_{\text{left}} \vee \mathtt{bb}_{\text{right}}\right) \wedge \left(\mathtt{bb}_{\text{exit}} \Rightarrow \mathtt{bb}_{\text{join}}\right) \wedge \mathtt{bb}_{\text{exit}} \\
\textbf{DEF} &= \left(\mathtt{bb}_{\text{entry}} \Rightarrow c = (x < y)\right) \wedge \left(\mathtt{bb}_{\text{left}} \Rightarrow a_{\text{left}} = x + 1\right) \\
&\quad \wedge \left(\mathtt{bb}_{\text{right}} \Rightarrow a_{\text{right}} = y + 1\right) \wedge \left(\mathtt{bb}_{\text{join}} \Rightarrow \mathrm{ok} = (a > 0)\right) \\
\textbf{JUMPS} &= \left(\left(\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{left}}\right) \Rightarrow c\right) \wedge \left(\left(\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{right}}\right) \Rightarrow \neg c\right) \\
\textbf{PHI} &= \left(\left(\mathtt{bb}_{\text{left}} \wedge \mathtt{bb}_{\text{join}}\right) \Rightarrow a = a_{\text{left}}\right) \wedge \left(\left(\mathtt{bb}_{\text{right}} \wedge \mathtt{bb}_{\text{join}}\right) \Rightarrow a = a_{\text{right}}\right) \\
\textbf{ERROR} &= \mathtt{bb}_{\text{exit}} \Rightarrow \top \wedge \neg\, \mathrm{ok}
\end{aligned}
$$

A satisfying model is a candidate unsafe execution: a selected entry-to-exit path, consistent assignments and edge conditions, and a false assertion at `exit`.

Optimization: Unguarded DEF

Static definitions may be exposed as top-level equations:

Guarded:

$$\mathsf{bb}_i \Rightarrow x = y + 1$$

Unguarded:

$$x = y + 1$$

Optimization: Unguarded DEF

Static definitions may be exposed as top-level equations:

Guarded:

$$\mathtt{bb}_i \Rightarrow x = y + 1$$

Unguarded:

$$x = y + 1$$

Why bother? Top-level equalities feed algebraic simplification:

$$x = y + 1 \land z = x + 2 \quad \Longrightarrow \quad z = y + 3$$

Sound only when the definition and its side conditions are globally safe — **keep that clause in mind for the pitfalls.**

Optimization: Phi As ITE

Edge implications:

$$(\mathsf{bb}_i \wedge \mathsf{bb}_j) \Rightarrow x = x_i$$

$$(\mathsf{bb}_k \wedge \mathsf{bb}_j) \Rightarrow x = x_k$$

One defining equation:

$$x = \mathrm{ite}(\mathsf{bb}_i, x_i, x_k)$$

After Boolean propagation learns bb_i, the ITE collapses to $x = x_i$.

The ITE form gives the merge a syntactic definition the solver can substitute — at a price we will meet in pitfall 2.

The Optimized Diamond

Same program — forward CFG, unguarded DEF, phi as ITE:

$$
\begin{aligned}
\textbf{CFG} &= \mathtt{bb}_{\text{entry}} \wedge \left(\mathtt{bb}_{\text{entry}} \Rightarrow \mathtt{bb}_{\text{left}} \vee \mathtt{bb}_{\text{right}}\right) \wedge \left(\mathtt{bb}_{\text{left}} \Rightarrow \mathtt{bb}_{\text{join}}\right) \\
&\quad \wedge \left(\mathtt{bb}_{\text{right}} \Rightarrow \mathtt{bb}_{\text{join}}\right) \wedge \left(\mathtt{bb}_{\text{join}} \Rightarrow \mathtt{bb}_{\text{exit}}\right) \wedge \mathtt{bb}_{\text{exit}} \\
\textbf{DEF} &= c = (x < y) \wedge a_{\text{left}} = x + 1 \wedge a_{\text{right}} = y + 1 \\
&\quad \wedge a = \mathrm{ite}\left(\mathtt{bb}_{\text{left}}, a_{\text{left}}, a_{\text{right}}\right) \wedge \mathrm{ok} = (a > 0) \\
\textbf{JUMPS} &= \left(\left(\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{left}}\right) \Rightarrow c\right) \wedge \left(\left(\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{right}}\right) \Rightarrow \neg c\right) \\
\textbf{PHI} &= \top \\
\textbf{ERROR} &= \mathtt{bb}_{\text{exit}} \Rightarrow \top \wedge \neg\ \mathrm{ok}
\end{aligned}
$$

Reachability flows forward; definitions are naked equations ready for substitution; the phi component is **empty** — the merge lives in DEF as an ITE.

Pitfall 1: A Critical Edge

```
entry:
  c := havoc
  if c goto join else mid

mid:
  goto join

join:
  ok := phi [entry: true, mid: false]
```

The encoder emits the true-edge condition:

$$\left(\mathsf{bb}_{\text{entry}} \wedge \mathsf{bb}_{\text{join}}\right) \Rightarrow c$$

The program is unsafe (take the `mid` path). Does the encoder find the bug?

Pitfall 1: A Critical Edge

```
entry:
  c := havoc
  if c goto join else mid

mid:
  goto join

join:
  ok := phi [entry: true, mid: false]
```

The encoder emits the true-edge condition:

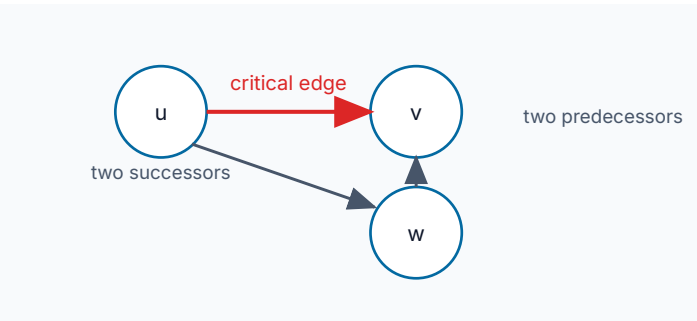
$$\left(\mathtt{bb}_{\mathrm{entry}} \wedge \mathtt{bb}_{\mathrm{join}}\right) \Rightarrow c$$

The program is unsafe (take the `mid` path). Does the encoder find the bug?

No. On `entry -> mid -> join`, both $\mathtt{bb}_{\mathrm{entry}}$ and $\mathtt{bb}_{\mathrm{join}}$ are true while c is false — the constraint **rejects the real bug path**. Block variables cannot tell which incoming edge was taken.

Pitfall 1: The Repair — Split The Edge

An edge $u \to v$ is critical when $|\mathrm{succ}(u)| > 1 \wedge |\mathrm{pred}(v)| > 1$:



Insert a landing block:

```
entry:
  c := havoc
  if c goto e2j else mid
e2j:
  goto join
```

The condition attaches to a non-critical edge:

$$\left(\mathsf{bb}_{\mathrm{entry}} \wedge \mathsf{bb}_{\mathrm{e2j}}\right) \Rightarrow c$$

$\mathsf{bb}_{\mathrm{e2j}}$ is true **only** on the true edge.

Pitfall 2: A Safe Program Goes SAT

```
left:
  x_left := true
  p_left := true
  goto join

right:
  x_right := false
  p_right := false
  goto join

join:
  x := phi [left: x_left, right: x_right]
  p := phi [left: p_left, right: p_right]
  ok := x == p
```

Safe: both paths make `x` and `p` equal.

The two phis are emitted as independent ITEs, cases in different orders:

$$x = \mathrm{ite}\left(\mathtt{bb}_{\mathrm{left}}, x_{\mathrm{left}}, x_{\mathrm{right}}\right)$$

$$p = \mathrm{ite}\left(\mathtt{bb}_{\mathrm{right}}, p_{\mathrm{right}}, p_{\mathrm{left}}\right)$$

The solver reports SAT. What model did it find?

Pitfall 2: A Safe Program Goes SAT

```
left:
  x_left := true
  p_left := true
  goto join

right:
  x_right := false
  p_right := false
  goto join

join:
  x := phi [left: x_left, right: x_right]
  p := phi [left: p_left, right: p_right]
  ok := x == p
```

Safe: both paths make `x` and `p` equal.

The two phis are emitted as independent ITEs, cases in different orders:

$$x = \mathrm{ite}\left(\mathtt{bb}_{\mathrm{left}}, x_{\mathrm{left}}, x_{\mathrm{right}}\right)$$

$$p = \mathrm{ite}\left(\mathtt{bb}_{\mathrm{right}}, p_{\mathrm{right}}, p_{\mathrm{left}}\right)$$

The solver reports SAT. What model did it find?

$\mathtt{bb}_{\mathrm{left}} = \mathtt{bb}_{\mathrm{right}} = \mathrm{true}$: the first ITE reads the **left** value, the second reads the **right** value — $x = \mathrm{true}, p = \mathrm{false}$. A mixed state no execution produces: a **spurious counterexample**.

Pitfall 2: The Repair — Predecessor Exclusivity

With edge implications, selecting both predecessors is an immediate contradiction ($x = x_{\text{left}}$ and $x = x_{\text{right}}$ conflict).

An ITE merge has only **one** equality — it silently picks an arm.

The repair is an at-most-one constraint at the merge:

$$\neg\left(\mathsf{bb}_{\text{left}} \wedge \mathsf{bb}_{\text{right}}\right)$$

Equivalently: any CFG encoding with edge variables must enforce that at most one incoming edge to a merge fires.

Phi-as-ITE trades a built-in exclusivity check for solver-friendliness — the side condition must come back explicitly.

Pitfall 3: An Unguarded Path-Local Fact

`narrow64` is identity, but its encoding adds a 64-bit range fact for its result:

```
entry:
  x := havoc
  small := x < 2^64
  if small goto narrow_block else exit

narrow_block:
  y := narrow64(x)
  goto exit

exit:
  assume not small
  assert false
  halt
```

Unsafe: pick $x \geq 2^{64}$, skip `narrow_block`, reach `assert false`.

The encoder emits the definition **unguarded**:

$$y = x \wedge 0 \leq y \wedge y \leq 2^{64} - 1$$

Does the solver find the bug?

Pitfall 3: An Unguarded Path-Local Fact

`narrow64` is identity, but its encoding adds a 64-bit range fact for its result:

```
entry:
  x := havoc
  small := x < 2^64
  if small goto narrow_block else exit

narrow_block:
  y := narrow64(x)
  goto exit

exit:
  assume not small
  assert false
  halt
```

Unsafe: pick $x \geq 2^{64}$, skip `narrow_block`, reach `assert false`.

The encoder emits the definition **unguarded**:

$$y = x \wedge 0 \leq y \wedge y \leq 2^{64} - 1$$

Does the solver find the bug?

No — UNSAT. The global range fact bounds x on **every** path; the failing path needs $x \geq 2^{64}$ and is now inconsistent. A fact from an unvisited block killed a real bug.

Pitfall 3: The Repair — Guard Path-Local Facts

Keep the range fact local to its block:

$$\texttt{bb}_{\text{narrow\_block}} \Rightarrow \left(y = x \wedge 0 \leq y \wedge y \leq 2^{64} - 1\right)$$

The unguarded-DEF optimization is safe for **total** right-hand sides. It breaks the moment the definition smuggles in a **path-local axiom** — a range fact, a partial-operation side condition, an operator axiom valid only where the call appears.

Guard such facts by their triggering block, or prove them globally safe.

The Two Ways To Be Wrong

Pitfall	Effect	Failure mode
critical edges	**missing paths** — a real execution is ruled out	bug silently missed (unsoundness)
ITE merge without exclusivity	**infeasible paths** — the formula admits states no execution produces	spurious counterexample (incompleteness)
unguarded path-local facts	**missing paths** — facts from an unvisited block constrain the failing path	bug silently missed (unsoundness)

Missing paths break **soundness**: the tool is silent on a real bug — a wrong "proved". Infeasible paths break **completeness**: noise — a false alarm on a safe program. Every encoding change should be audited against both directions.

VCGen

Boogie Style

Idea

Boogie-style VCGen uses backward safety equations over the CFG.

For each block `i`, introduce:

$$\mathrm{ok}_i$$

`ok_i` means: starting execution at block `i` is safe.

The counterexample query is:

$$\neg \mathrm{ok}_{\mathrm{entry}}$$

No path selection, no block-reachability variables — control flow becomes recursion.

Block Equation

$$\mathrm{ok}_i \Leftrightarrow \left(\mathrm{defs}_i \Rightarrow \left(\mathrm{asserts}_i \wedge \wedge_{j \in \mathrm{succ}(i)} \left(\left(\mathrm{cond}_{i,j} \wedge \mathrm{dsa}_{i,j}\right) \Rightarrow \mathrm{ok}_j\right)\right)\right)$$

- assumptions and definitions are premises
- assertions are consequents
- each feasible successor must be safe
- infeasible blocks are safe vacuously

Terminal Block

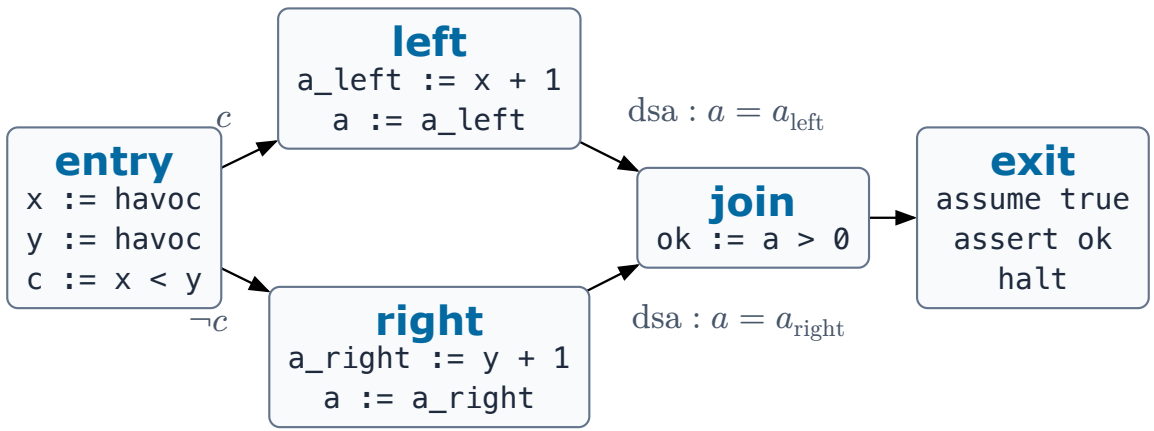
For a terminal block, the successor conjunction is `true`:

$$\mathrm{ok}_i \Leftrightarrow (\mathrm{defs}_i \Rightarrow \mathrm{asserts}_i)$$

A false reachable assertion makes the local `ok_i` false, then unsafety propagates backward to `entry`.

The Diamond, In DSA Form

Boogie-style consumes DSA: the merge moves onto the incoming edges.



The assignments to `a` are **edge-sensitive DSA definitions** — they enter the formula on the edge premises, not as block facts.

Encoded Equations

$$\mathbf{OK}_{\text{entry}} : \text{ok}_{\text{entry}} \Leftrightarrow \big(c = (x < y) \Rightarrow \big((c \Rightarrow \text{ok}_{\text{left}}) \wedge (\neg c \Rightarrow \text{ok}_{\text{right}})\big)\big)$$

$$\mathbf{OK}_{\text{left}} : \text{ok}_{\text{left}} \Leftrightarrow \big(a_{\text{left}} = x + 1 \Rightarrow \big((a = a_{\text{left}}) \Rightarrow \text{ok}_{\text{join}}\big)\big)$$

$$\mathbf{OK}_{\text{right}} : \text{ok}_{\text{right}} \Leftrightarrow \big(a_{\text{right}} = y + 1 \Rightarrow \big((a = a_{\text{right}}) \Rightarrow \text{ok}_{\text{join}}\big)\big)$$

Encoded Exit

$$\mathbf{OK}_{\text{join}} : \text{ok}_{\text{join}} \Leftrightarrow (\text{ok} = (a > 0) \Rightarrow \text{ok}_{\text{exit}})$$

$$\mathbf{OK}_{\text{exit}} : \text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$$

$$\mathbf{DISCHARGE} : \neg\text{ok}_{\text{entry}}$$

The model chooses havoc values and a branch whose successor is unsafe.

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$
- OK_{exit}: $\text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$, so $\text{ok}_{\text{exit}} = \text{false}$

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$
- OK_{exit}: $\text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$, so $\text{ok}_{\text{exit}} = \text{false}$
- OK_{join}: premise $\text{ok} = (a > 0)$ holds ($-1 > 0$ is false), successor unsafe $\longrightarrow \text{ok}_{\text{join}} = \text{false}$

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$
- OK_{exit}: $\text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$, so $\text{ok}_{\text{exit}} = \text{false}$
- OK_{join}: premise $\text{ok} = (a > 0)$ holds ($-1 > 0$ is false), successor unsafe $\longrightarrow \text{ok}_{\text{join}} = \text{false}$
- OK_{left}: premises $a_{\text{left}} = x + 1 = -1$ and edge definition $a = a_{\text{left}}$ hold $\longrightarrow \text{ok}_{\text{left}} = \text{false}$

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$
- OK_{exit}: $\text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$, so $\text{ok}_{\text{exit}} = \text{false}$
- OK_{join}: premise $\text{ok} = (a > 0)$ holds ($-1 > 0$ is false), successor unsafe $\longrightarrow \text{ok}_{\text{join}} = \text{false}$
- OK_{left}: premises $a_{\text{left}} = x + 1 = -1$ and edge definition $a = a_{\text{left}}$ hold $\longrightarrow \text{ok}_{\text{left}} = \text{false}$
- OK_{entry}: $c = (x < y) = \text{true}$, and the true branch requires $c \Rightarrow \text{ok}_{\text{left}} \longrightarrow \text{ok}_{\text{entry}} = \text{false}$

A Model, Walked Backward

The diamond is unsafe (a can be ≤ 0). Watch the solver's model flow through the equations:

- model picks $x = -2, \quad y = 0, \quad a = a_{\text{left}} = -1, \quad \text{ok} = \text{false}$
- OK_{exit}: $\text{ok}_{\text{exit}} \Leftrightarrow \text{ok}$, so $\text{ok}_{\text{exit}} = \text{false}$
- OK_{join}: premise $\text{ok} = (a > 0)$ holds ($-1 > 0$ is false), successor unsafe $\longrightarrow \text{ok}_{\text{join}} = \text{false}$
- OK_{left}: premises $a_{\text{left}} = x + 1 = -1$ and edge definition $a = a_{\text{left}}$ hold $\longrightarrow \text{ok}_{\text{left}} = \text{false}$
- OK_{entry}: $c = (x < y) = \text{true}$, and the true branch requires $c \Rightarrow \text{ok}_{\text{left}} \longrightarrow \text{ok}_{\text{entry}} = \text{false}$
- **DISCHARGE** $\neg\text{ok}_{\text{entry}}$ is satisfied — the model **is** the failing execution `entry -> left -> join -> exit`.

Static ITE Merge Variant

The merge can be compiled into an ordinary definition:

```
join:
  a := ite(c, a_left, a_right)
  ok := a > 0
  goto exit
```

$$\mathrm{ok}_{\mathrm{join}} \Leftrightarrow \Big(\big(a = \mathrm{ite}\big(c, a_{\mathrm{left}}, a_{\mathrm{right}}\big) \wedge \mathrm{ok} = (a > 0)\big) \Rightarrow \mathrm{ok}_{\mathrm{exit}}\Big)$$

Now `left` and `right` have no edge definitions — the predecessor-sensitive merge became a local fact of `join`.

Tradeoffs

Advantages:

- no separate path-selection formula
- many assertions are natural
- assumptions are handled by implication

Costs:

- formula is nested
- cross-block simplification is harder
- phi/ITE extensions need exclusivity side conditions

Phi Extension Needs CFG Facts

If a phi is compiled as:

$$a = \mathrm{ite}\big(\mathtt{bb}_{\mathrm{left}}, a_{\mathrm{left}}, a_{\mathrm{right}}\big)$$

then the block variables used by the ITE must be consistent:

$$\mathtt{bb}_{\mathrm{join}} \Rightarrow \mathtt{bb}_{\mathrm{left}} \vee \mathtt{bb}_{\mathrm{right}}$$

$$\neg\big(\mathtt{bb}_{\mathrm{left}} \wedge \mathtt{bb}_{\mathrm{right}}\big)$$

The backward equations prove safety; the CFG facts keep the ITE merge faithful — the same exclusivity pitfall as SeaHorn's phi-as-ITE.

Same Diamond, Two Formulas

SeaHorn — flat conjunction:

$$\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{exit}} \wedge \ldots$$
$$\mathtt{bb}_{\text{entry}} \Rightarrow c = (x < y)$$
$$\mathtt{bb}_{\text{left}} \Rightarrow a_{\text{left}} = x + 1$$
$$\left(\mathtt{bb}_{\text{entry}} \wedge \mathtt{bb}_{\text{left}}\right) \Rightarrow c$$
$$\left(\mathtt{bb}_{\text{left}} \wedge \mathtt{bb}_{\text{join}}\right) \Rightarrow a = a_{\text{left}}$$
$$\mathtt{bb}_{\text{exit}} \Rightarrow \neg\ \mathrm{ok}$$

facts at top level; path selected by block variables

Boogie — nested recursion:

$$\neg \mathrm{ok}_{\text{entry}}$$
$$\mathrm{ok}_{\text{entry}} \Leftrightarrow \big(c = (x < y) \Rightarrow$$
$$\big((c \Rightarrow \mathrm{ok}_{\text{left}}) \wedge \big(\neg c \Rightarrow \mathrm{ok}_{\text{right}}\big)\big)\big)$$
$$\mathrm{ok}_{\text{left}} \Leftrightarrow \big(a_{\text{left}} = x + 1 \Rightarrow$$
$$\big((a = a_{\text{left}}) \Rightarrow \mathrm{ok}_{\text{join}}\big)\big)$$
$$\ldots$$

facts nested under equations; path selected by implication chains

Same models, same verdict — different levers for the solver: SeaHorn exposes equalities for substitution; Boogie gets assertions and assumptions for free.

VCGen

SeaBMC Style

Idea

SeaBMC-style VCGen compiles control-dependence into data-dependence.

After preprocessing:

- no block variables are needed
- no separate CFG constraints are emitted
- phi nodes become ordinary `ite` definitions
- cone-of-influence reduction is just data-flow slicing

SeaHorn kept the CFG in the formula; Boogie compiled it into recursion; SeaBMC makes it **disappear**.

Gated SSA

A gamma node is a value-level merge:

$$x := \gamma\big(g, x_t, x_f\big)$$

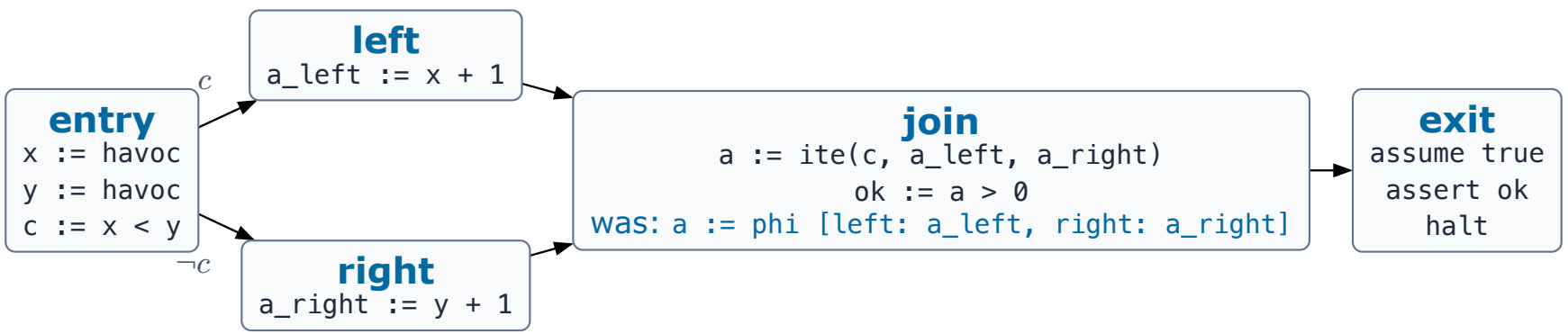
It means:

$$x := \mathrm{ite}\big(g, x_t, x_f\big)$$

Unlike a phi node, `gamma` mentions a Boolean **program expression**, not a predecessor block name.

From Phi To Gamma

The branch condition `c` already decides the incoming region — let the merge switch on **it**:



`a := gamma(c, a_left, a_right)` desugars to `a := ite(c, a_left, a_right)` — an ordinary SSA definition. No CFG predicate needed to explain the merge.

Flat VC

After phi elimination:

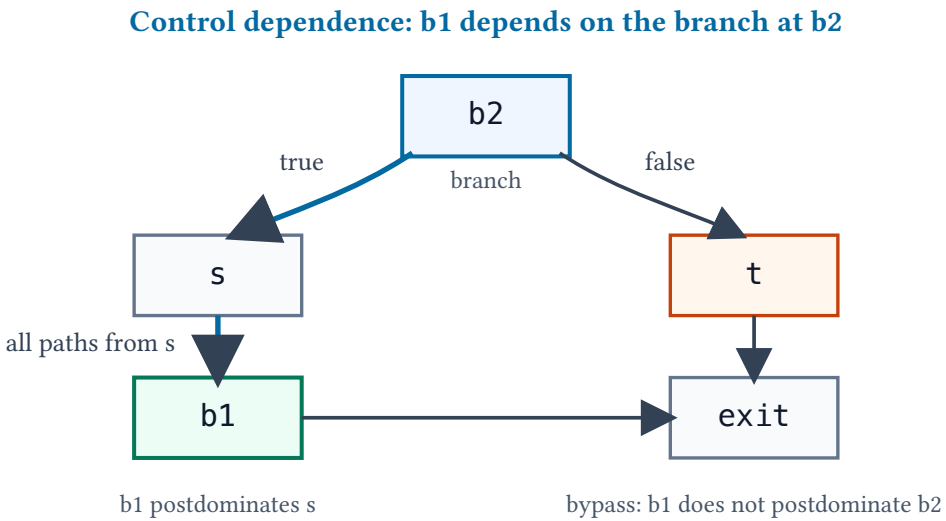
$$\mathrm{VCGen\_SeaBMC}(P) = \mathbf{DEF} \wedge \mathbf{ASSUME} \wedge \neg\, \mathbf{ASSERT}$$

For the diamond:

$$\mathbf{DEF} = c = (x < y) \wedge a_{\text{left}} = x + 1 \wedge a_{\text{right}} = y + 1 \wedge a = \mathrm{ite}\big(c, a_{\text{left}}, a_{\text{right}}\big) \wedge \mathrm{ok} = (a > 0)$$

No block variables, no CFG constraints. The only trace of control flow is the ITE guard in the definition of `a`.

Control Dependence



- d **postdominates** b: every path $b \to$ `exit` passes through d.
- b_1 is **control-dependent** on branch b_2: some successor of b_2 forces a later visit to b_1, while b_1 does not postdominate b_2 itself — the branch genuinely decides.

Gate Construction

For each block b, compute a Boolean expression `gate(b)` over **program** branch conditions:

$$\begin{aligned} \mathrm{gate}(\mathrm{entry}) &= \top \\ \mathrm{gate}(b) &= \bigvee_{c \in \mathrm{ctrl}(b)} \left(\mathrm{gate}(c) \wedge \mathrm{orient}(c, b)\right) \end{aligned}$$

- `ctrl(b)`: branch blocks b is control-dependent on (empty $\longrightarrow \mathrm{gate}(b) = \top$).
- `orient(c,b)`: the branch condition of c, or its negation, depending on which successor reaches b.

In a structured diamond: $\mathrm{gate}(\mathrm{left}) = c$, $\mathrm{gate}(\mathrm{right}) = \neg c$.

The Hidden Blow-Up

`gate` is recursive. Substituted textually into every use, shared control structure is **copied**:

$$\begin{aligned}
\mathrm{gate}(c) &= (p \wedge r) \vee (q \wedge s) \\
\mathrm{gate}(d) &= (q \wedge t) \vee (((p \wedge r) \vee (q \wedge s)) \wedge u) \\
\mathrm{gate}(e) &= ((((p \wedge r) \vee (q \wedge s)) \wedge v) \vee (((q \wedge t) \vee (((p \wedge r) \vee (q \wedge s)) \wedge u)) \wedge w)
\end{aligned}$$

Every level repeats its ancestors; every gamma that uses $\mathrm{gate}(e)$ repeats the whole tree again — **exponential** in the DAG depth.

Materialized Gates

Name each gate as an ordinary Boolean SSA definition, and let gammas refer to the names:

$$G_{\text{entry}} := \top$$
$$G_b := \bigvee_{c \in \text{ctrl}(b)} (G_c \wedge \text{orient}(c, b))$$

For the blow-up example:

$$G_c := (G_a \wedge r) \vee (G_b \wedge s) \qquad G_d := (G_b \wedge t) \vee (G_c \wedge u) \qquad G_e := (G_c \wedge v) \vee (G_d \wedge w)$$

The DAG is represented once — **linear** in gates plus uses. The VC stays CFG-free: gates are just definitions.

Materialized Gates In The Program

Gates are just definitions; the VC stays CFG-free:

```
gate_a := p
gate_b := q
gate_c := (gate_a and r) or (gate_b and s)
gate_d := (gate_b and t) or (gate_c and u)
gate_e := (gate_c and v) or (gate_d and w)

join1:
  x := ite(gate_d, x_d, ite(gate_e, x_e, x_other))

join2:
  y := ite(gate_d, y_d, ite(gate_e, y_e, y_other))
```

`join1` and `join2` share the same control-dependence DAG — it appears once, not once per merge.

Thin GSSA

Materialization stops the copying, but gates can still enumerate every path from `entry`. **Thin GSSA** switches only on the **direct** control-dependence controllers:

$$K_{c,b} = G_c \wedge \mathrm{orient}(c, b)$$

Why the G_c factor? SSA variables are **total** in the formula:

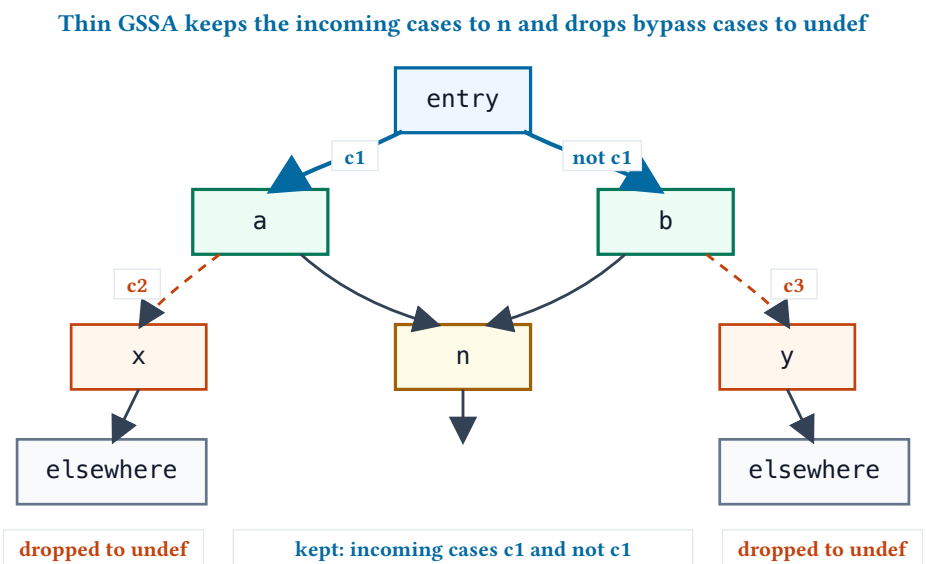
> If controller block c is never reached, its branch condition still has a model value. Switching on the condition alone lets an **unreachable** branch select the merge value.

G_c says "controller c itself is reached" — dead branch conditions cannot fire a case.

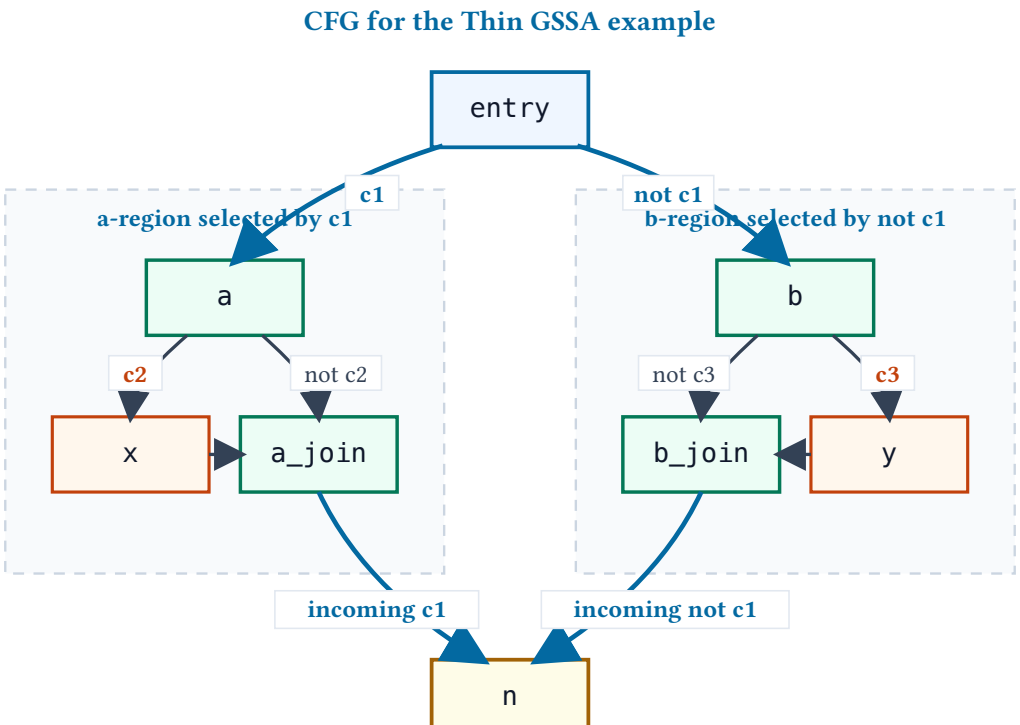
The Fallback: false And undef

If no direct-controller case fires, execution does not reach b:

- For the **reachability gate** G_b, the fallback is `false` — not reached means not reached.
- For a **value** merge in b, the fallback is `undef`: a fresh, unconstrained symbol of the right type. The value is unobservable when b is not reached — any value will do.



Thin GSSA, Derived



The outer branch `c1` selects the region feeding `n`; the local branches `c2`, `c3` only compute values **inside** each region.

Thin GSSA, Derived: Three Steps

Step 1 — regular GSSA: each phi uses the full path condition of its predecessor:

```
a_join:
  v_a := gamma(c1 and c2, v_x, gamma(c1 and not c2, v_a0, undef))

b_join:
  v_b := gamma(not c1 and c3, v_y,
         gamma(not c1 and not c3, v_b0, undef))

n:
  v := gamma(c1, v_a, gamma(not c1, v_b, undef))
```

Every gate repeats the outer condition `c1`.

Thin GSSA, Derived: Three Steps

Step 2 — drop dead fallbacks: the program is SESE, so execution reaching a join arrives from exactly one predecessor — complete two-way choices never select `undef`:

```
a_join:
  v_a := gamma(c2, v_x, gamma(not c2, v_a0, undef))

b_join:
  v_b := gamma(c3, v_y, gamma(not c3, v_b0, undef))

n:
  v := gamma(c1, v_a, v_b)
```

The local gates no longer mention `c1` — the **direct** controller is enough.

Thin GSSA, Derived: Three Steps

Step 3 — thin form, after collapsing the two-way choices and desugaring gamma:

```
a_join:
  v_a := ite(c2, v_x, v_a0)

b_join:
  v_b := ite(c3, v_y, v_b0)

n:
  v := ite(c1, v_a, v_b)
```

The final merge at `n` depends **only** on the region selector `c1`; each local ITE depends only on its own branch. Path conditions never materialize.

Cone Of Influence

Once control is data:

1. Start from the exit `assume` and `assert`.
2. Keep each used variable definition.
3. Add the variables used by that definition.
4. Keep gates only when retained ITEs use them.
5. Drop every unmarked definition.

No graph reachability rule is needed for pruning.

COI Example

```
n:
  v := ite(c1, v_a, v_b)
  junk := expensive(v_x, v_y)
  ok := v > 0

exit:
  assume true
  assert ok
```

`junk` is dropped because it is not used by `ok`, the exit assumption, or a retained gate.

A branch that gates no retained value simply vanishes from the formula.

Three Encodings, One Diamond

	SeaHorn (03)	**Boogie** (04)	**SeaBMC** (05)
control flow	explicit: $\texttt{bb}_i$ variables + path constraints	recursion: backward ok_i equations	compiled away: gates as data
program form	SSA	DSA	SSA → gated SSA
joins	PHI edge implications / ITE	DSA edge definitions	gamma over branch conditions
formula shape	flat conjunction	nested implications	flat conjunction, CFG-free
solver lever	top-level equalities, Boolean propagation on $\texttt{bb}_i$	assertions and assumes for free	substitution and slicing everywhere
watch out for	critical edges, merge exclusivity, unguarded facts	exclusivity for phi/ITE extensions	gate blow-up without materialization

ctac's default encoder (`sea_vc`) is SeaHorn-style — block variables over DSA — with the guarded/unguarded and merge-shape choices exposed as flags.

VC Generation

References And Borrowing

Goal

Extend Tiny TAC with references without adding a new VCGen core.

Plan:

- say what borrow commands **mean** as executions
- find the one place where that meaning fights SSA
- solve it with a value chosen from the future — a **prophecy**
- lower borrows to ordinary Tiny TAC and reuse the existing VCGen

The Puzzle This Deck Solves

```
entry:
  M := havoc
  i := havoc
  r, M2 := borrow_mut M[i]
  r2 := put_ref r, 7
  release r2
  x := M2[i]
  ok := x == 7
  assert ok
  halt
```

`borrow_mut` returns the reference **and** the continuation memory `M2` — in SSA, before any write through `r` has happened.

`x` is read from `M2`, which was created **before** the `7` was written. Why should `assert ok` hold?

By the end of the deck this will be obvious.

Reference Types

Tiny TAC gains:

$$\tau ::= \text{bool} \mid \text{int} \mid \text{bytemap} \mid \& \ \tau \mid \&\text{mut} \ \tau$$

The main case is a reference to an integer cell:

$$r : \&\text{int} \quad q : \&\text{mut int}$$

Borrow Commands

```
r := borrow M[i]
r, M2 := borrow_mut M[i]
q := borrow_ref r
q, r2 := borrow_ref_mut r
x := get_ref r
r2 := put_ref r, v
release r
```

Mutable borrowing returns both the reference and a continuation memory. Mutable reborrowing returns the child reference and the resumed parent reference.

That is the syntax. Before any encoding: what do these commands **do**?

What Borrowing Means

Read `r, M2 := borrow_mut M[i]` as a **loan** of slot `i`:

- while the loan lives, `r` is the only way to touch slot `i`
- `get_ref r` reads the loaned view; `put_ref r, v` overwrites it (returning a fresh reference, to stay in SSA)
- `release` ends the loan: the **last value written** through the reference is committed back to memory
- `M2` names the memory **after the loan ends** — the rest of the program reads memory through `M2`

> `M2` is **named** at borrow time, but its content at `i` is **determined** at release time.

That temporal gap is the entire difficulty of this deck.

The Puzzle, Executed

Run the loan semantics forward over the puzzle program:

after command	view through the borrow	M2[i]
`r, M2 := borrow_mut M[i]`	`r` sees `M[i]`	**?** — committed at release
`r2 := put_ref r, 7`	`r2` sees `7`	**?**
`release r2`	loan ends, `7` is committed	`7`
`x := M2[i]`	—	`7`, so `x = 7` and `ok` holds

Execution is fine: time flows forward, and the **?** resolves before anyone reads it. But an SSA definition of `M2` cannot wait — it needs a value for slot `i` **at borrow time**.

A Guess That Must Come True

Forget references. Tiny TAC can already talk about the future:

```
entry:
  guess := havoc
  double := guess * 2
  answer := 7
  assume guess == answer
  ok := double == 14
  assert ok
```

`double` is computed from `guess` **before** `answer` exists.

Is `assert ok` provable? `guess` is an arbitrary integer…

A Guess That Must Come True

Forget references. Tiny TAC can already talk about the future:

```
entry:
  guess := havoc
  double := guess * 2
  answer := 7
  assume guess == answer
  ok := double == 14
  assert ok
```

`double` is computed from `guess` **before** `answer` exists.

Is `assert ok` provable? `guess` is an arbitrary integer…

Yes — UNSAT. The `assume` discards every run whose guess was wrong. In all surviving runs `guess = 7`, so `double = 14` — as if `double` had been computed from the future value all along.

Why The Guess Trick Is Sound

- **havoc** = fork one run per possible future value of `guess`
- **assume** = at the moment the future arrives, kill every run where the guess disagrees
- for each real execution, **exactly one** guess survives — the correct one

So the program with (`havoc` now + `assume` later) has **the same observable behaviors** as one where the value magically arrived early: no behavior lost, none invented.

This is a **prophecy variable**. The formula is timeless — a constraint placed late reaches every use placed early. Only forward execution finds prophecies strange (it would have to guess).

Reference Triple

Now the borrow encoding writes itself. A reference is a triple:

$$r = \{\text{addr} : i, \text{value} : v, \text{promise} : p\}$$

- `addr`: the loaned location
- `value`: what the reference **currently sees**
- `promise`: the prophecy — a guess of the value the loan will commit at release

`promise` is exactly the `guess` from the previous slide, one per borrow: havoc'd when the loan starts, settled when it ends. Constant references never use theirs, but one shape keeps the lowering uniform.

Lowering: Direct Borrows

Constant borrow:

```
r := borrow M[i]
```

lowers to:

```
r := { addr: i,
       value: M[i],
       promise: havoc }
```

Mutable borrow:

```
r, M2 := borrow_mut M[i]
```

lowers to:

```
r := { addr: i,
       value: M[i],
       promise: havoc }
M2 := M[i := r.promise]
```

The continuation memory is defined **now**, at slot `i`, with the prophecy — the SSA gap is closed by the guess.

Lowering: Use And Release

Read:

```
x := get_ref r
```

lowers to:

```
x := r.value
```

Write:

```
r2 := put_ref r, v
```

lowers to:

```
r2 := { addr: r.addr,
        value: v,
        promise: r.promise }
```

Release:

```
assume r.value == r.promise
```

`release` is the `assume` of the guess trick: the value observed at the end of the loan **is** the promised value.

Original And Desugared, Side By Side

Original:

```
entry:
  M := havoc
  i := havoc
  r, M2 := borrow_mut M[i]
  r2 := put_ref r, 7
  release r2
  x := M2[i]
  ok := x == 7
  assert ok
```

Desugared:

```
entry:
  M := havoc
  i := havoc
  r := { addr: i, value: M[i],
         promise: havoc }
  M2 := M[i := r.promise]
  r2 := { addr: r.addr, value: 7,
          promise: r.promise }
  assume r2.value == r2.promise
  x := M2[i]
  ok := x == 7
  assert ok
```

The borrow becomes three facts: observe the current value, havoc the promise, and store the promise into the continuation memory — the future is written into `M2` on day one.

Original And Desugared, Side By Side

Original:

```
entry:
  M := havoc
  i := havoc
  r, M2 := borrow_mut M[i]
  r2 := put_ref r, 7
  release r2
  x := M2[i]
  ok := x == 7
  assert ok
```

Desugared:

```
entry:
  M := havoc
  i := havoc
  r := { addr: i, value: M[i],
         promise: havoc }
  M2 := M[i := r.promise]
  r2 := { addr: r.addr, value: 7,
          promise: r.promise }
  assume r2.value == r2.promise
  x := M2[i]
  ok := x == 7
  assert ok
```

The write produces a fresh view whose `value` is `7`. The promise rides along unchanged — the guess is about the **final** committed value, whoever writes it.

Original And Desugared, Side By Side

Original:

```
entry:
  M := havoc
  i := havoc
  r, M2 := borrow_mut M[i]
  r2 := put_ref r, 7
  release r2
  x := M2[i]
  ok := x == 7
  assert ok
```

Desugared:

```
entry:
  M := havoc
  i := havoc
  r := { addr: i, value: M[i],
         promise: havoc }
  M2 := M[i := r.promise]
  r2 := { addr: r.addr, value: 7,
          promise: r.promise }
  assume r2.value == r2.promise
  x := M2[i]
  ok := x == 7
  assert ok
```

The release settles the guess: the last observed value equals the promise. From here on, every run that survives has `r.promise = 7`.

Original And Desugared, Side By Side

Original:

```
entry:
  M := havoc
  i := havoc
  r, M2 := borrow_mut M[i]
  r2 := put_ref r, 7
  release r2
  x := M2[i]
  ok := x == 7
  assert ok
```

Desugared:

```
entry:
  M := havoc
  i := havoc
  r := { addr: i, value: M[i],
         promise: havoc }
  M2 := M[i := r.promise]
  r2 := { addr: r.addr, value: 7,
          promise: r.promise }
  assume r2.value == r2.promise
  x := M2[i]
  ok := x == 7
  assert ok
```

The tail is untouched. `M2[i]` already contains the — now settled — promise, so `x = 7`. Only ordinary assignments, a store, and an assume remain: existing VCGen applies unchanged.

Why It Proves

$$\begin{aligned} \mathrm{M2} &= M[i := \mathrm{promise}(r)] \\ \mathrm{value}(\mathrm{r2}) &= 7 \\ \mathrm{promise}(\mathrm{r2}) &= \mathrm{promise}(r) \\ \mathrm{value}(\mathrm{r2}) &= \mathrm{promise}(\mathrm{r2}) \end{aligned}$$

Why It Proves

$$\begin{aligned} \mathrm{M2} &= M[i := \mathrm{promise}(r)] \\ \mathrm{value}(\mathrm{r2}) &= 7 \\ \mathrm{promise}(\mathrm{r2}) &= \mathrm{promise}(r) \\ \mathrm{value}(\mathrm{r2}) &= \mathrm{promise}(\mathrm{r2}) \end{aligned}$$

Chaining the last three equalities:

$$\mathrm{promise}(r) = 7$$

So `M2` — written back at borrow time — stores `7` at address `i`, and `x == 7` follows. The havoc'd promise was **forced** by the release assumption.

Demo: The Whole Pipeline

Desugaring is a `ttac -> ttac` pass; the VC core never sees a reference:

```
$ ttac desugar safe_borrow_mut.ttac | ttac vcgen - --solve
unsat
$ ttac desugar unsafe_borrow_mut.ttac | ttac vcgen - --solve
sat
```

Demo: The Whole Pipeline

Desugaring is a `ttac -> ttac` pass; the VC core never sees a reference:

```
$ ttac desugar safe_borrow_mut.ttac | ttac vcgen - --solve
unsat
$ ttac desugar unsafe_borrow_mut.ttac | ttac vcgen - --solve
sat
```

And the prediction from the guess-trick slide — forward execution **cannot** run a borrow program:

```
$ ttac run safe_borrow_mut.ttac
status: stopped (assume failed in block entry)
```

The interpreter picks a value for the havoc'd promise and dies at the release `assume`. Only the solver — choosing all values at once — threads the prophecy.

Mutable Reborrow

```
r, M2 := borrow_mut M[i]
q, r2 := borrow_ref_mut r
q2 := put_ref q, 7
release q2
r3 := put_ref r2, 8
release r3
```

`q` borrows through `r`: a loan of a loan. While `q` lives, `r` is suspended; after `q` is released, `r2` resumes the parent reference and can overwrite the final promised value.

Reborrow Lowering

```
q := { addr: r.addr, value: r.value, promise: havoc }
r2 := { addr: r.addr, value: q.promise, promise: r.promise }
```

Two prophecies now: `q.promise` guesses what the **child** loan commits; the parent's resumed view `r2` starts at exactly that guess. `r2` keeps the original promise — the outer loan still owes memory its final value.

Releasing `q` settles the inner guess; releasing `r2` settles the outer one.

Reborrow Facts

$$\begin{aligned}
\mathrm{M2} &= M[i := \mathrm{promise}(r)] \\
\mathrm{value}(\mathrm{q2}) &= 7 \\
\mathrm{value}(\mathrm{q2}) &= \mathrm{promise}(q) \\
\mathrm{value}(\mathrm{r3}) &= 8 \\
\mathrm{promise}(\mathrm{r3}) &= \mathrm{promise}(\mathrm{r2}) \\
\mathrm{promise}(\mathrm{r2}) &= \mathrm{promise}(r)
\end{aligned}$$

The inner release forces `promise(q) = 7`, so `r2` resumes seeing `7`. The outer release forces `promise(r) = 8` — the parent's final write wins, and `M2[i] == 8`.

Prophecy Variables: The Name And The History

What we built has a name and a literature:

- Abadi and Lamport, *The Existence of Refinement Mappings* (1991) — prophecy variables: auxiliary values chosen nondeterministically now, constrained by the future, without changing observable behavior.
- Matsushita, Tsukada, Kobayashi, *RustHorn: CHC-based Verification for Rust Programs* (2020) — a mutable borrow **is** a (current, final) value pair; no memory model needed for safe Rust.
- Priya and Gurfinkel, *Ownership in low-level intermediate representation* (FMCAD 2024) — the same idea in a BMC setting, mixed with an address-map memory model. Tiny TAC follows this pattern.

The `promise` field is RustHorn's "final value", made to coexist with explicit bytemaps: the borrow writes it into the continuation memory, the release proves it right.

VCGen Boundary

After lowering, VCGen only sees ordinary Tiny TAC:

- assignments
- map reads and stores
- assumptions
- assertions
- havoc values

Borrow **well-formedness** (liveness, exclusivity, reborrow lifetimes) is a separate side condition — checked before VCGen, or encoded as extra obligations. Candidate disciplines: Stacked Borrows, Tree Borrows.

Open Extensions

- **Finite-region borrows** — borrow a byte range, not a single cell: base + size + per-word value/promise. Question: keeping the encoding compact.
- **Reference shadow memory** — store references in memory via ghostmap shadows for addr/value/promise/permission. Question: which metadata must be shadowed, and how shadows stay synchronized.
- **Reference semantics and well-formedness** — pick a borrow model (Stacked/Tree Borrows), state the rules, decide: reject before VCGen or encode as obligations.

Each is small enough to state independently, but substantial enough to be a useful project.