Challenges (and Solutions) in Memory-Model-Aware Verification

Levente Bajczi

September 6, 2024

ftsrg Critical Systems Research Group

Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$\mathbf{x} := 1$	$a := \mathbf{x}$
$\mathbf{x} := 2$	$b := \mathbf{x}$

Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & a := \mathbf{x} \\ \mathbf{x} := 2 & b := \mathbf{x} \end{array}$$

po
Instruction order

co
Memory update order (W -> W)

rf
Dataflow (W -> R)

$$\forall r^{\mathbf{x}} \exists w^{\mathbf{x}} : w^{\mathbf{x}} \xrightarrow{rf} r^{\mathbf{x}}$$

$$\forall w_1^{\mathbf{x}}, w_2^{\mathbf{x}} : w_1^{\mathbf{x}} \xrightarrow{co} w_2^{\mathbf{x}} \vee w_2^{\mathbf{x}} \xrightarrow{co} w_1^{\mathbf{x}}$$

Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$\mathbf{x} := 1$	$a := \mathbf{x}$
$\mathbf{x} := 2$	$b := \mathbf{x}$

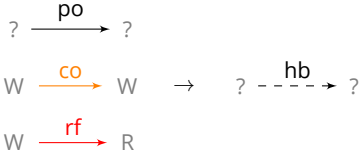
po — Instruction order

co — Memory update order (W -> W)

rf — Dataflow (W -> R)

$$\forall r^{\mathbf{x}} \exists w^{\mathbf{x}} : w^{\mathbf{x}} \xrightarrow{rf} r^{\mathbf{x}}$$
$$\forall w_1^{\mathbf{x}}, w_2^{\mathbf{x}} : w_1^{\mathbf{x}} \xrightarrow{co} w_2^{\mathbf{x}} \vee w_2^{\mathbf{x}} \xrightarrow{co} w_1^{\mathbf{x}}$$

Memory model (example)



Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$$\mathbf{x} := 1 \mid a := \mathbf{x}$$
$$\mathbf{x} := 2 \mid b := \mathbf{x}$$

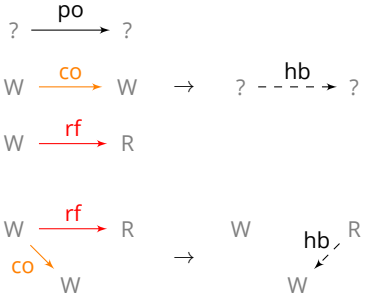
po — Instruction order

co — Memory update order (W -> W)

rf — Dataflow (W -> R)

$$\forall r^{\mathbf{x}} \exists w^{\mathbf{x}} : w^{\mathbf{x}} \xrightarrow{rf} r^{\mathbf{x}}$$
$$\forall w_1^{\mathbf{x}}, w_2^{\mathbf{x}} : w_1^{\mathbf{x}} \xrightarrow{co} w_2^{\mathbf{x}} \vee w_2^{\mathbf{x}} \xrightarrow{co} w_1^{\mathbf{x}}$$

Memory model (example)



Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$$\mathbf{x} := 1 \mid a := \mathbf{x}$$
$$\mathbf{x} := 2 \mid b := \mathbf{x}$$

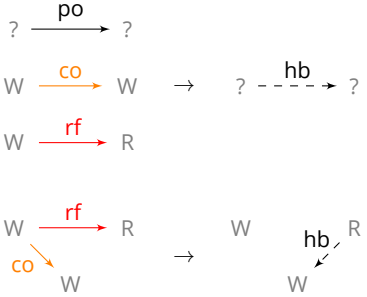
po — Instruction order

co — Memory update order (W -> W)

rf — Dataflow (W -> R)

$$\forall r^{\mathbf{x}} \exists w^{\mathbf{x}} : w^{\mathbf{x}} \xrightarrow{rf} r^{\mathbf{x}}$$
$$\forall w_1^{\mathbf{x}}, w_2^{\mathbf{x}} : w_1^{\mathbf{x}} \xrightarrow{co} w_2^{\mathbf{x}} \vee w_2^{\mathbf{x}} \xrightarrow{co} w_1^{\mathbf{x}}$$

Memory model (example)



Find *co* and *rf* for a given *po* such that ***hb*** is acyclic.

Challenges for Memory-Model-Aware Verification

Program

$$\mathbf{x} := 0$$

$$\mathbf{x} := 1 \quad \Big| \quad a := \mathbf{x}$$
$$\mathbf{x} := 2 \quad \Big| \quad b := \mathbf{x}$$

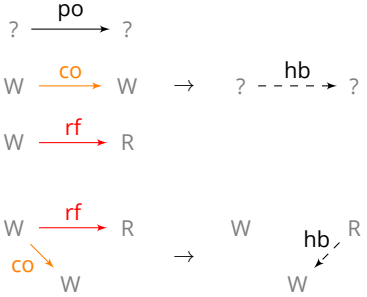
po — Instruction order

co — Memory update order (W -> W)

rf — Dataflow (W -> R)

$$\forall r^{\mathbf{x}} \exists w^{\mathbf{x}} : w^{\mathbf{x}} \xrightarrow{rf} r^{\mathbf{x}}$$
$$\forall w_1^{\mathbf{x}}, w_2^{\mathbf{x}} : w_1^{\mathbf{x}} \xrightarrow{co} w_2^{\mathbf{x}} \vee w_2^{\mathbf{x}} \xrightarrow{co} w_1^{\mathbf{x}}$$

Memory model (example)



Witnesses?

Safety?

Tools?

Find *co* and *rf* for a given *po* such that ***hb*** is acyclic.

Memory-Model-Aware Verification

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

Program with property

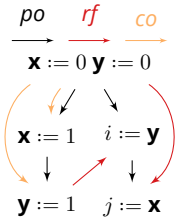
Memory-Model-Aware Verification

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$\mathbf{x} := 1$	$i := \mathbf{y}$
$\mathbf{y} := 1$	$j := \mathbf{x}$

$$\neg(i = 1 \wedge j = 0)$$

Program with property



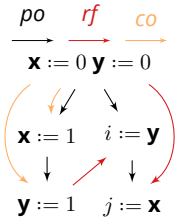
CEx. candidate(s) *(obeying axioms)*

Memory-Model-Aware Verification

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$\mathbf{x} := 1$	$i := \mathbf{y}$
$\mathbf{y} := 1$	$j := \mathbf{x}$

$$\neg(i = 1 \wedge j = 0)$$



Program with property

CEx. candidate(s) *(obeying axioms)*

Sequential Consistency

All instructions execute in-order

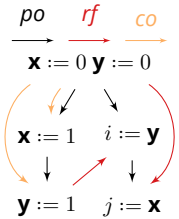
Safe (inconsistent)

Memory-Model-Aware Verification

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$



Program with property

CEx. candidate(s) *(obeying axioms)*

Sequential Consistency

All instructions execute in-order

Safe (inconsistent)

Total Store Order

Indep. W->R order is not obeyed

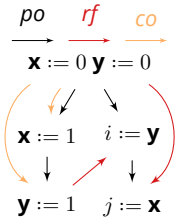
Safe (inconsistent)

Memory-Model-Aware Verification

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$



Program with property

CEx. candidate(s) *(obeying axioms)*

Sequential Consistency

All instructions execute in-order

Safe (inconsistent)

Total Store Order

Indep. W->R order is not obeyed

Safe (inconsistent)

Partial Store Order

Indep. W->? order is not obeyed

Unsafe (consistent)

State-of-the-Art Tools

Exhaustive Enumeration

- Generate execution candidates, and check their consistency

Herd7

(memory model simulator)

- Litmus tests
- CAT memory model

State-of-the-Art Tools

Exhaustive Enumeration

- Generate execution candidates, and check their consistency

Herd7

(memory model simulator)

- Litmus tests
- CAT memory model

Stateless Model Checking

- Generate increasingly larger, always consistent executions (traces)

GenMC, **Nidhugg**, …

- (Subset of) C11
- Custom library / hardcoded

State-of-the-Art Tools

Exhaustive Enumeration

- Generate execution candidates, and check their consistency

Herd7

(memory model simulator)

- Litmus tests
- CAT memory model

Stateless Model Checking

- Generate increasingly larger, always consistent executions (traces)

GenMC, **Nidhugg**, …

- (Subset of) C11
- Custom library / hardcoded

Bounded Model Checking

- Encode constraints of the memory model in the SMT query

Dartagnan

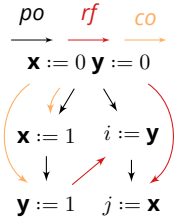
- (SV-COMP flavored) C
- Subset of CAT

State-of-the-Art Tools

Exhaustive Enumeration	Stateless Model Checking	Bounded Model Checking
▶ Generate execution candidates, and check their consistency	▶ Generate increasingly larger, always consistent executions (traces)	▶ Encode constraints of the memory model in the SMT query
Herd7 (memory model simulator)	**GenMC**, **Nidhugg**, …	**Dartagnan**
▶ Litmus tests	▶ (Subset of) C11	▶ (SV-COMP flavored) C
▶ CAT memory model	▶ Custom library / hardcoded	▶ Subset of CAT

- (Almost) no interoperability
- Verdicts are not verifiable *(Dartagnan does produce witnesses for SC)*

Witnesses for Violations (in Witness Format 2!)



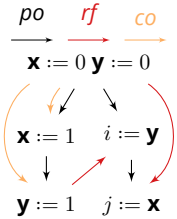
PSO

Indep. W->? order is not obeyed

Unsafe (consistent)

Witnesses for Violations (in Witness Format 2!)

Thread 0	waypoint type	value	line	column
Thread 1				
Thread 2				



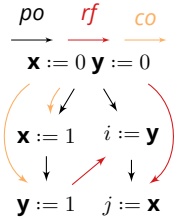
PSO

Indep. W->? order is not obeyed

Unsafe (consistent)

Witnesses for Violations (in Witness Format 2!)

Thread 0	waypoint type	value	line	column
Thread 1	thread_start	1, 2	1	0
Thread 2				
	target	-	2	*end*



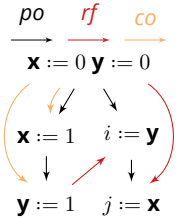
PSO

Indep. W->? order is not obeyed

Unsafe (consistent)

Witnesses for Violations (in Witness Format 2!)

Thread 0	waypoint type	value	line	column
	assume	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
	assume	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
	thread_start	1, 2	1	0
Thread 1				
Thread 2				
	target	-	2	*end*



PSO

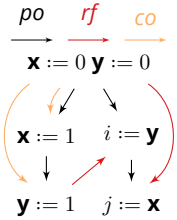
Indep. W->? order is not obeyed

Unsafe (consistent)

- $\backslash at(\mathbf{e}, \mathbf{id})$: Built-in ACSL construct *(abused a bit)*
  - *[...] referring to the value of the expression **e** in the state at label **id*** [ACSL 1.20]
  - Our *state labels* are integers, and denote ordering of memory events.

Witnesses for Violations (in Witness Format 2!)

Thread 0	waypoint type	value	line	column
	assume	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
	assume	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
	thread_start	1, 2	1	0
Thread 1				
	assume	$\backslash at(\mathbf{x}, 1) = 1$	1	*end*
	assume	$\backslash at(\mathbf{y}, 1) = 1$	2	*end*
Thread 2				
	target	-	2	*end*



PSO

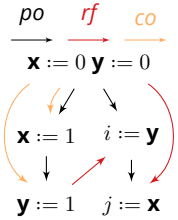
Indep. W->? order is not obeyed

Unsafe (consistent)

- $\backslash at(\mathbf{e}, \mathbf{id})$: Built-in ACSL construct *(abused a bit)*
  - *[...] referring to the value of the expression **e** in the state at label **id*** [ACSL 1.20]
  - Our *state labels* are integers, and denote ordering of memory events.

Witnesses for Violations (in Witness Format 2!)

Thread 0	waypoint type	value	line	column
	assume	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
	assume	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
	thread_start	1, 2	1	0
Thread 1				
	assume	$\backslash at(\mathbf{x}, 1) = 1$	1	*end*
	assume	$\backslash at(\mathbf{y}, 1) = 1$	2	*end*
Thread 2				
	assume	$i = \backslash at(\mathbf{x}, 1)$	1	*end*
	assume	$j = \backslash at(\mathbf{y}, 0)$	2	*end*
	target	-	2	*end*



PSO

Indep. W->? order is not obeyed

Unsafe (consistent)

- $\backslash at(\mathbf{e}, \mathbf{id})$: Built-in ACSL construct *(abused a bit)*
  - *[...] referring to the value of the expression **e** in the state at label **id*** [ACSL 1.20]
  - Our *state labels* are integers, and denote ordering of memory events.

Witnesses for Correctness

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$\mathbf{x} := 1$	$i := \mathbf{y}$
$\mathbf{y} := 1$	$j := \mathbf{x}$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

Witnesses for Correctness

invariant type	value	line	column

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

Witnesses for Correctness

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

Witnesses for Correctness

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*
location	$\exists a : a \in \{0, 1\}$ $i = \backslash at(\mathbf{x}, a)$	1 *(right)*	*end*

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

- *state labels* are **symbolic** integers, and denote ordering of memory events.

Witnesses for Correctness

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*
location	$\exists a : a \in \{0, 1\}$ $i = \backslash at(\mathbf{x}, a)$	1 *(right)*	*end*
location	$\exists a, b : a, b \in \{0, 1\}$ $j = \backslash at(\mathbf{y}, a)$ $i = \backslash at(\mathbf{x}, b)$ $b = 1 \implies a = 1$	2 *(right)*	*end*

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$\mathbf{x} := 1$	$i := \mathbf{y}$
$\mathbf{y} := 1$	$j := \mathbf{x}$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

- *state labels* are **symbolic** integers, and denote ordering of memory events.

Witnesses for Correctness

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*
location	$\exists a : a \in \{0, 1\}$ $i = \backslash at(\mathbf{x}, a)$	1 *(right)*	*end*
location	$\exists a, b : a, b \in \{0, 1\}$ $j = \backslash at(\mathbf{y}, a)$ $i = \backslash at(\mathbf{x}, b)$ $b = 1 \implies a = 1$	2 *(right)*	*end*
location	$\neg(i = 1 \wedge j = 0)$	2 *(right)*	*end*

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

SC

All instructions execute in-order

Safe (inconsistent)

- *state labels* are **symbolic** integers, and denote ordering of memory events.

Witnesses for Correctness

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*
location	$\exists a : a \in \{0, 1\}$ $i = \backslash at(\mathbf{x}, a)$	1 *(right)*	*end*
location	$\exists a, b : a, b \in \{0, 1\}$ $j = \backslash at(\mathbf{y}, a)$ $i = \backslash at(\mathbf{x}, b)$ $b = 1 \implies a = 1$	2 *(right)*	*end*
location	$\neg(i = 1 \wedge j = 0)$	2 *(right)*	*end*

$$\mathbf{x} := 0, \mathbf{y} := 0$$

$$\begin{array}{l|l} \mathbf{x} := 1 & i := \mathbf{y} \\ \mathbf{y} := 1 & j := \mathbf{x} \end{array}$$

$$\neg(i = 1 \wedge j = 0)$$

SC

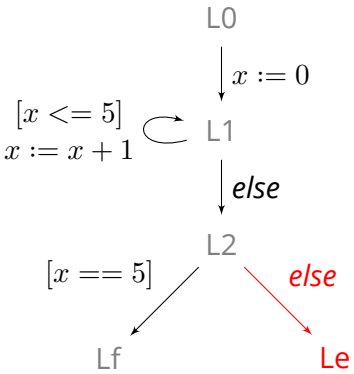
All instructions execute in-order

Safe (inconsistent)

- *state labels* are **symbolic** integers, and denote ordering of memory events.

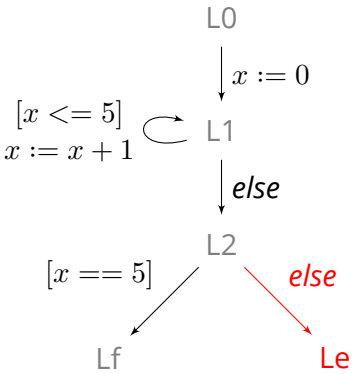
Where do invariants come from?

CHC Crash Course




```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

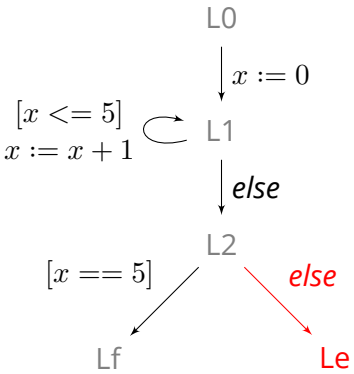
CHC Crash Course



$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

CHC Crash Course

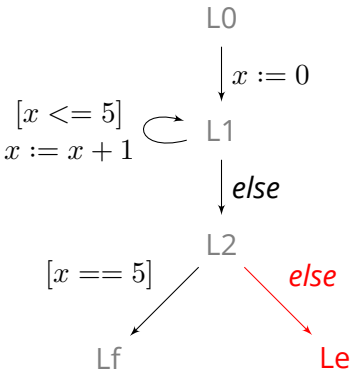



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$L_0(x) \quad \longleftarrow \quad \top$$

CHC Crash Course



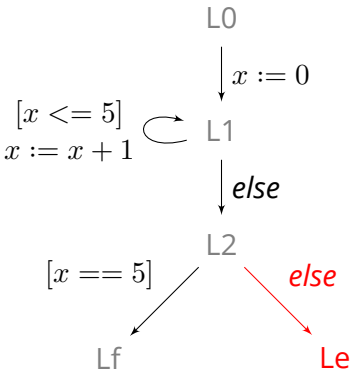

```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$L_0(x) \longleftarrow \top$$

$$\forall x : L_0(x)$$

CHC Crash Course

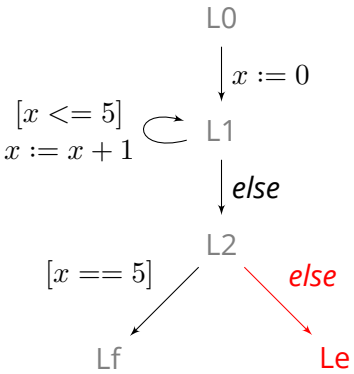



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{aligned} L_0(x) &\longleftarrow \top \\ & \qquad \forall x : L_0(x) \\ L_1(x') &\longleftarrow L_0(x) \wedge x' = 0 \end{aligned}$$

CHC Crash Course

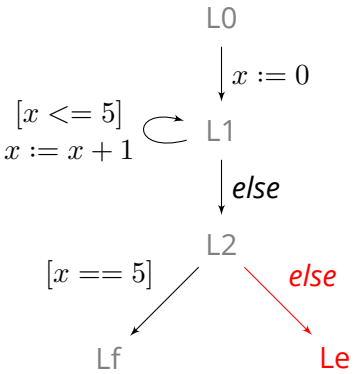



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{aligned} L_0(x) &\longleftarrow \top \\ &\qquad \forall x : L_0(x) \\ L_1(x') &\longleftarrow L_0(x) \wedge x' = 0 \\ &\qquad L_1(0) \end{aligned}$$

CHC Crash Course

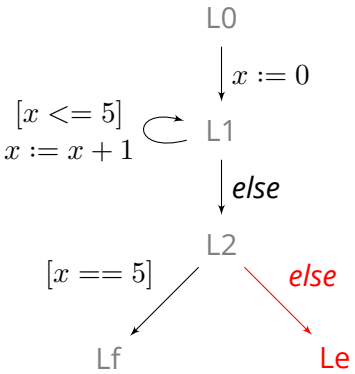



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \end{array}$$

CHC Crash Course

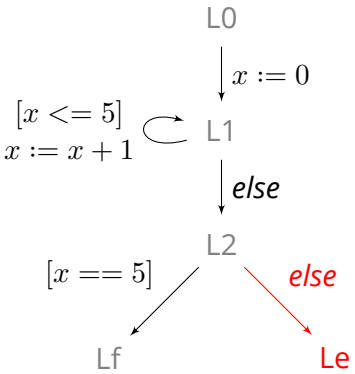



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), \end{array}$$

CHC Crash Course

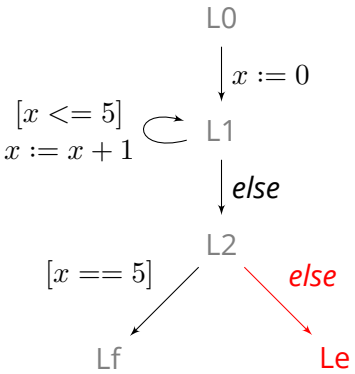



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), L_1(2), \end{array}$$

CHC Crash Course

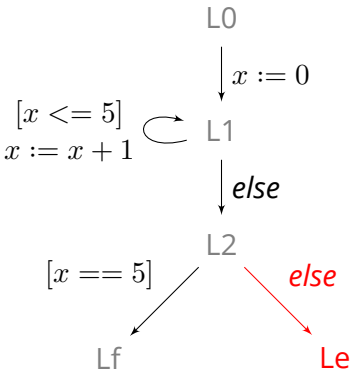



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), L_1(2), L_1(3), \end{array}$$

CHC Crash Course

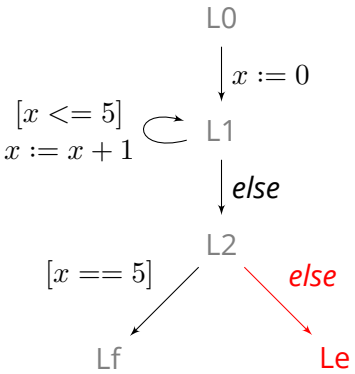



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), L_1(2), L_1(3), L_1(4), \end{array}$$

CHC Crash Course

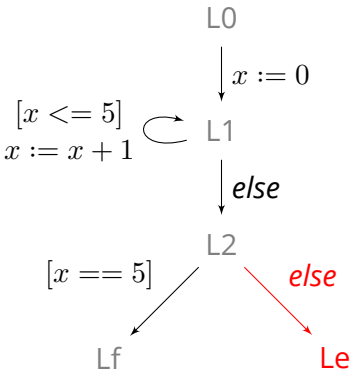



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), L_1(2), L_1(3), L_1(4), L_1(5), \end{array}$$

CHC Crash Course

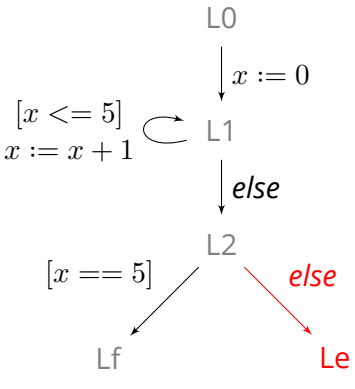



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$\begin{array}{lcl} L_0(x) & \longleftarrow & \top \\ & & \forall x : L_0(x) \\ L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\ & & L_1(0) \\ L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\ & & L_2(1), L_1(2), L_1(3), L_1(4), L_1(5), L_1(6) \end{array}$$

CHC Crash Course

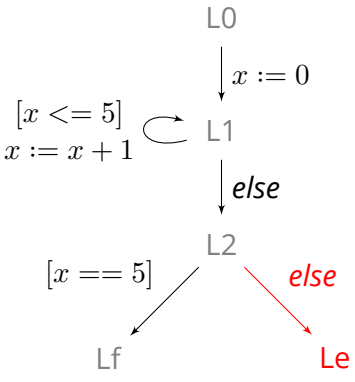



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$
\begin{array}{lcl}
L_0(x) & \longleftarrow & \top \\
 & & \forall x : L_0(x) \\
L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\
 & & L_1(0) \\
L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\
 & & L_2(1), L_1(2), L_1(3), L_1(4), L_1(5), L_1(6) \\
L_2(x) & \longleftarrow & L_1(x) \wedge !(x \leq 5)
\end{array}
$$

CHC Crash Course

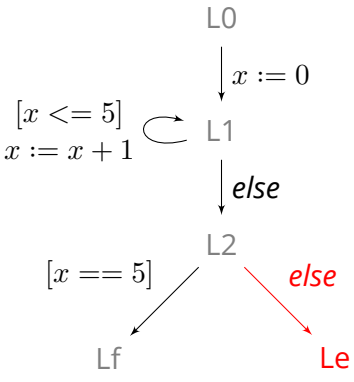



```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$
\begin{array}{lcl}
L_0(x) & \longleftarrow & \top \\
 & & \forall x : L_0(x) \\
L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\
 & & L_1(0) \\
L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\
 & & L_2(1), L_1(2), L_1(3), L_1(4), L_1(5), L_1(6) \\
L_2(x) & \longleftarrow & L_1(x) \wedge !(x \leq 5) \\
 & & L_2(6)
\end{array}
$$

CHC Crash Course




```
int x = 0;
while(x <= 5)
    x++;
assert(x == 5);
```

$$\mathit{TARGET}(vars') \longleftarrow \mathit{SRC}(vars) \wedge edge(vars, vars')$$

$$
\begin{array}{lcl}
L_0(x) & \longleftarrow & \top \\
 & & \forall x : L_0(x) \\
L_1(x') & \longleftarrow & L_0(x) \wedge x' = 0 \\
 & & L_1(0) \\
L_1(x') & \longleftarrow & L_1(x) \wedge x \leq 5 \wedge x' = x + 1 \\
 & & L_2(1), L_1(2), L_1(3), L_1(4), L_1(5), L_1(6) \\
L_2(x) & \longleftarrow & L_1(x) \wedge !(x \leq 5) \\
 & & L_2(6) \\
\bot & \longleftarrow & L_2(x) \wedge !(x = 5)
\end{array}
$$

Constrained Horn Clauses for Weak Memory (Level 1)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do $\{$	*do* $\{$
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	$\}$*while*$(a \wedge b)$	$\}$*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

Constrained Horn Clauses for Weak Memory (Level 1)

$\mathbf{flag_0} := 0, \mathbf{flag_1} := 0, \mathbf{turn} := 0, \mathbf{cnt} := 0$

L_0	$\mathbf{flag_0} := 1$	$\mathbf{flag_1} := 1$
L_1	$\mathbf{turn} := 1$	$\mathbf{turn} := 0$
	do $\{$	*do* $\{$
L_2	$a := \mathbf{flag_1}$	$a := \mathbf{flag_0}$
L_3	$b := \mathbf{turn}$	$b := \mathbf{turn}$
L_4	$\}$*while*$(a \wedge b)$	$\}$*while*$(a \wedge \neg b)$
C_0	$c := \mathbf{cnt}$	$c := \mathbf{cnt}$
C_1	$\mathbf{cnt} := c + 1$	$\mathbf{cnt} := c + 1$
C_2	$\mathbf{cnt} := 0$	$\mathbf{cnt} := 0$
L_5	$\mathbf{flag_0} := 0$	$\mathbf{flag_1} := 0$

- Encode threads independently
  - Write: *NOP*
  - Read: *havoc*

Constrained Horn Clauses for Weak Memory (Level 1)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do {	*do* {
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	}*while*$(a \wedge b)$	}*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

- Encode threads independently
  - Write: *NOP*
  - Read: *havoc*

$$
\begin{array}{lcll}
L_0^{T_0}(a,b,c) & \longleftarrow & \top & \text{//init} \\
L_1^{T_0}(a,b,c) & \longleftarrow & L_0^{T_0}(a,b,c) & \text{//skip} \\
L_2^{T_0}(a,b,c) & \longleftarrow & L_1^{T_0}(a,b,c) & \text{//skip} \\
L_3^{T_0}(a',b,c) & \longleftarrow & L_2^{T_0}(a,b,c) & \text{//havoc } a \\
 & & \ldots & \\
L_0^{T_0}(a,b,c) & \longleftarrow & L_5^{T_0}(a,b,c) & \text{//repeat} \\
\bot & \longleftarrow & C_1^{T_0}(a,b,c) \wedge \neg(c=0) & \text{//error}
\end{array}
$$

Constrained Horn Clauses for Weak Memory (Level 1)

$\mathbf{flag_0} := 0, \mathbf{flag_1} := 0, \mathbf{turn} := 0, \mathbf{cnt} := 0$

L_0	$\mathbf{flag_0} := 1$	$\mathbf{flag_1} := 1$
L_1	$\mathbf{turn} := 1$	$\mathbf{turn} := 0$
	do $\{$	*do* $\{$
L_2	$a := \mathbf{flag_1}$	$a := \mathbf{flag_0}$
L_3	$b := \mathbf{turn}$	$b := \mathbf{turn}$
L_4	$\}$*while*$(a \wedge b)$	$\}$*while*$(a \wedge \neg b)$
C_0	$c := \mathbf{cnt}$	$c := \mathbf{cnt}$
C_1	$\mathbf{cnt} := c + 1$	$\mathbf{cnt} := c + 1$
C_2	$\mathbf{cnt} := 0$	$\mathbf{cnt} := 0$
L_5	$\mathbf{flag_0} := 0$	$\mathbf{flag_1} := 0$

SAT: There is a safety proof.

- Encode threads independently
  - Write: *NOP*
  - Read: *havoc*

$$
\begin{array}{lcll}
L_0^{T_0}(a,b,c) & \longleftarrow & \top & \text{//init} \\
L_1^{T_0}(a,b,c) & \longleftarrow & L_0^{T_0}(a,b,c) & \text{//skip} \\
L_2^{T_0}(a,b,c) & \longleftarrow & L_1^{T_0}(a,b,c) & \text{//skip} \\
L_3^{T_0}(a',b,c) & \longleftarrow & L_2^{T_0}(a,b,c) & \text{//havoc } a \\
 & & \ldots & \\
L_0^{T_0}(a,b,c) & \longleftarrow & L_5^{T_0}(a,b,c) & \text{//repeat} \\
\bot & \longleftarrow & C_1^{T_0}(a,b,c) \wedge \neg(c=0) & \text{//error}
\end{array}
$$

Constrained Horn Clauses for Weak Memory (Level 1)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do {	*do* {
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	}*while*$(a \wedge b)$	}*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

- Encode threads independently
  - Write: *NOP*
  - Read: *havoc*

$$
\begin{array}{lcll}
L_0^{T_0}(a,b,c) & \longleftarrow & \top & \text{//init} \\
L_1^{T_0}(a,b,c) & \longleftarrow & L_0^{T_0}(a,b,c) & \text{//skip} \\
L_2^{T_0}(a,b,c) & \longleftarrow & L_1^{T_0}(a,b,c) & \text{//skip} \\
L_3^{T_0}(a',b,c) & \longleftarrow & L_2^{T_0}(a,b,c) & \text{//havoc } a \\
 & & \ldots & \\
L_0^{T_0}(a,b,c) & \longleftarrow & L_5^{T_0}(a,b,c) & \text{//repeat} \\
\bot & \longleftarrow & C_1^{T_0}(a,b,c) \wedge \neg(c=0) & \text{//error}
\end{array}
$$

SAT: There is a safety proof.

UNSAT: *May* be a counterexample.

Constrained Horn Clauses for Weak Memory (Level 2)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do $\{$	*do* $\{$
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	$\}$*while*$(a \wedge b)$	$\}$*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

- No temporal ordering
  - Write: entails $W(var, value)$
  - Read: asserts $W(var, value)$

$$W(0,0) \quad \longleftarrow \quad L_0^{T_0}(a,b,c) \;//\textbf{flag}_0 := 1$$

$$L_4^{T_0}(a,b',c) \quad \longleftarrow \quad L_3^{T_0}(a,b,c) \wedge W(2,b') \;//b := \textbf{turn}$$

Constrained Horn Clauses for Weak Memory (Level 2)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do {	*do* {
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	}*while*$(a \wedge b)$	}*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

- No temporal ordering
  - Write: entails $W(var, value)$
  - Read: asserts $W(var, value)$

$$W(0,0) \quad \leftarrow \quad L_0^{T_0}(a,b,c) \;//\textbf{flag}_0 := 1$$

$$L_4^{T_0}(a,b',c) \quad \leftarrow \quad L_3^{T_0}(a,b,c) \wedge W(2,b') \;//b := \textbf{turn}$$

SAT: There is a safety proof.

UNSAT: *May* be a counterexample.

Constrained Horn Clauses for Weak Memory (Level 3)

$\textbf{flag}_0 := 0, \textbf{flag}_1 := 0, \textbf{turn} := 0, \textbf{cnt} := 0$

L_0	$\textbf{flag}_0 := 1$	$\textbf{flag}_1 := 1$
L_1	$\textbf{turn} := 1$	$\textbf{turn} := 0$
	do {	*do* {
L_2	$a := \textbf{flag}_1$	$a := \textbf{flag}_0$
L_3	$b := \textbf{turn}$	$b := \textbf{turn}$
L_4	}*while*$(a \wedge b)$	}*while*$(a \wedge \neg b)$
C_0	$c := \textbf{cnt}$	$c := \textbf{cnt}$
C_1	$\textbf{cnt} := c + 1$	$\textbf{cnt} := c + 1$
C_2	$\textbf{cnt} := 0$	$\textbf{cnt} := 0$
L_5	$\textbf{flag}_0 := 0$	$\textbf{flag}_1 := 0$

- Single temporal ordering
  - Write: entails $W(var, value)$
  - Read: asserts $W(var, value)$
  - Every predicate: ordering metadata

$$
\begin{array}{lcl}
W(\prec, \prec_{co}, 0, 0, 0) & \longleftarrow & \top \\
l_0(\prec, \prec_{co}, 1) & \longleftarrow & \top \\
W(\prec, \prec_{co}, e, 0, 1) & \longleftarrow & l_0(\prec, \prec_{co}, e) \wedge \\
 & & W(\prec, \prec_{co}, e', 0, \_) \wedge \\
 & & \prec_{co}[e'] + 1 = \prec_{co}[e] \wedge \\
 & & \prec[e'] \prec \prec[e] \\
l_1(\prec, \prec_{co}, e) & \longleftarrow & l_0(\prec, \prec_{co}, e') \wedge \\
 & & W(\prec, \prec_{co}, e', \_, \_) \wedge \\
 & & e' + 1 = e
\end{array}
$$

Constrained Horn Clauses for Weak Memory (Level 3)

- $\mathit{SRC}(vars) \land edge(vars, vars') \implies \mathit{TARGET}(vars')$
- $\mathit{SRC}(vars, \mathbf{ord}) \land edge(vars, vars') \land \mathbf{consistent}(\mathbf{ord}, \dots) \implies T.(vars', \mathbf{ord})$

Constrained Horn Clauses for Weak Memory (Level 3)

- $\mathit{SRC}(vars) \wedge edge(vars, vars') \implies \mathit{TARGET}(vars')$
- $\mathit{SRC}(vars, \mathbf{ord}) \wedge edge(vars, vars') \wedge \mathbf{consistent}(\mathbf{ord}, \dots) \implies T.(vars', \mathbf{ord})$
  - the current edge can be added without causing new inconsistencies

Constrained Horn Clauses for Weak Memory (Level 3)

- $\mathit{SRC}(vars) \wedge edge(vars, vars') \implies \mathit{TARGET}(vars')$
- $\mathit{SRC}(vars, \mathbf{ord}) \wedge edge(vars, vars') \wedge \mathbf{consistent}(\mathbf{ord}, \dots) \implies T.(vars', \mathbf{ord})$
  - the current edge can be added without causing new inconsistencies

SAT: There is a safety proof.

UNSAT: *Must* be a counterexample.

Constrained Horn Clauses for Weak Memory (Level 3)

- $SRC(vars) \wedge edge(vars, vars') \implies TARGET(vars')$
- $SRC(vars, \mathbf{ord}) \wedge edge(vars, vars') \wedge \mathbf{consistent}(\mathbf{ord}, \dots) \implies T.(vars', \mathbf{ord})$
  - the current edge can be added without causing new inconsistencies

SAT: There is a safety proof.

UNSAT: *Must* be a counterexample.

- Proof-of-concept implementation: Thorn (Theta + HORN)
- Full Peterson example: CHCs for Weak Memory peterson.smt2

Thank you, Gidon!

Constrained Horn Clauses for Weak Memory (Level 3)

- $\mathit{SRC}(vars) \wedge edge(vars, vars') \implies \mathit{TARGET}(vars')$
- $\mathit{SRC}(vars, \textbf{ord}) \wedge edge(vars, vars') \wedge \textbf{consistent}(\textbf{ord}, \dots) \implies \mathit{T}.(vars', \textbf{ord})$
  - the current edge can be added without causing new inconsistencies

SAT: There is a safety proof.

UNSAT: *Must* be a counterexample.

- Proof-of-concept implementation: Thorn (Theta + HORN)
- Full Peterson example: CHCs for Weak Memory peterson.smt2
  Thank you, Gidon!
- Witnesses proposal: Software Verification Witnesses for Weak Memory
  Thank you, Marian!