Unbounded Model Checking: IC3 and PDR

Automated Program Verification (APV)
Fall 2019

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Project proposals due November 18, 2019

Talk to me before submitting the proposal!

Submit PDF with proposal by email

- Must include at least 3 references to be read during the project

SAT-based Model Checking

Bounded Model Checking

- Is there a counterexample of k-steps

Unbounded Model Checking

- Induction and K-Induction (k-IND)
- Interpolation Based Model Checking (IMC)
- Property Directed Reachability (IC3/PDR)

Symbolic Safety and Reachability

A transition system P = (V, Init, Tr, Bad)

P is UNSAFE if and only if there exists a number N s.t.

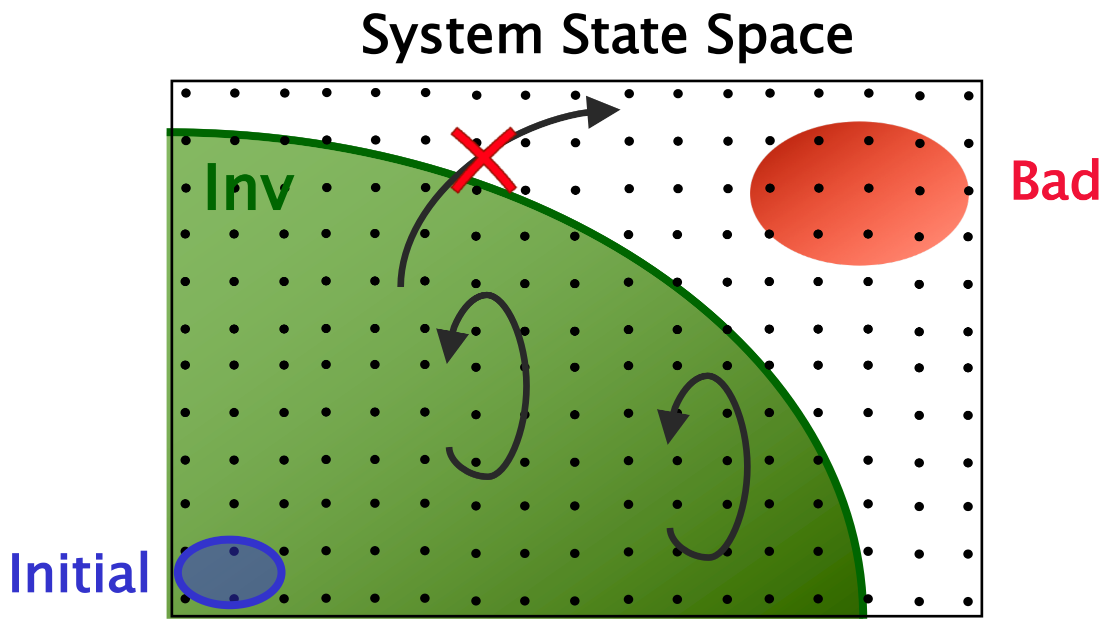
$$Init(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1})\right) \wedge Bad(X_N) \not\Rightarrow \bot$$

P is SAFE if and only if there exists a safe inductive invariant Inv s.t.

$$\left.\begin{array}{r}Init \Rightarrow Inv \\ Inv(X) \wedge Tr(X, X') \Rightarrow Inv(X')\end{array}\right\} \text{Inductive}$$

$$Inv \Rightarrow \neg Bad \quad \text{Safe}$$

Inductive Invariants

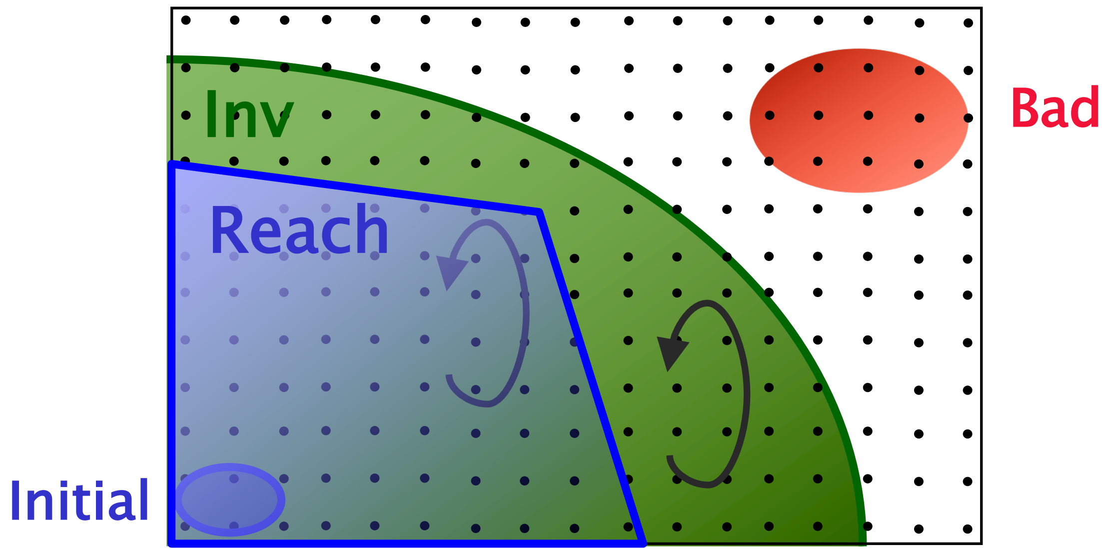


System S is safe iff there exists an inductive invariant Inv:

- **Initiation:** Initial $\subseteq$ Inv
- **Safety:** Inv $\cap$ Bad = $\emptyset$
- **Consecution:** TR(Inv) $\subseteq$ Inv

i.e., if s $\in$ Inv and s$\rightsquigarrow$t then t $\in$ Inv

Inductive Invariants



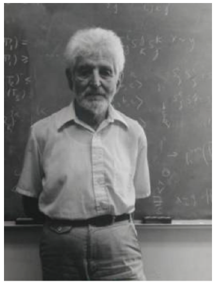
System S is safe iff there exists an inductive invariant Inv:

- **Initiation:** Initial ⊆ Inv
- **Safety:** Inv ∩ Bad = ∅
- **Consecution:** TR(Inv) ⊆ Inv

i.e., if s ∈ Inv and s⤳t then t ∈ Inv

System S is safe if Reach ∩ Bad = ∅

Craig Interpolants [Craig 57]



Given a pair (A,B) of propositional formulas s.t.

- $A(X,Y) \wedge B(Y,Z)$ is unsatisfiable
- i.e., $A \Rightarrow \neg B$

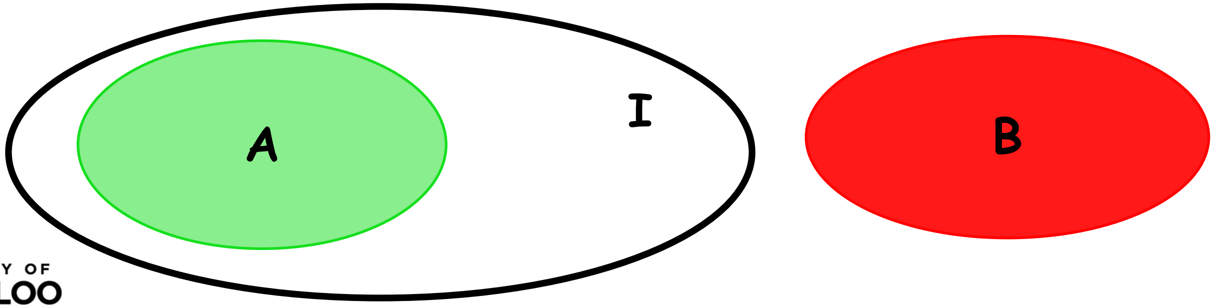
There exists a formula I such that:

- $A \Rightarrow I$
- $I \wedge B$ is unsatisfiable
- I is over Y, the common variables of A and B

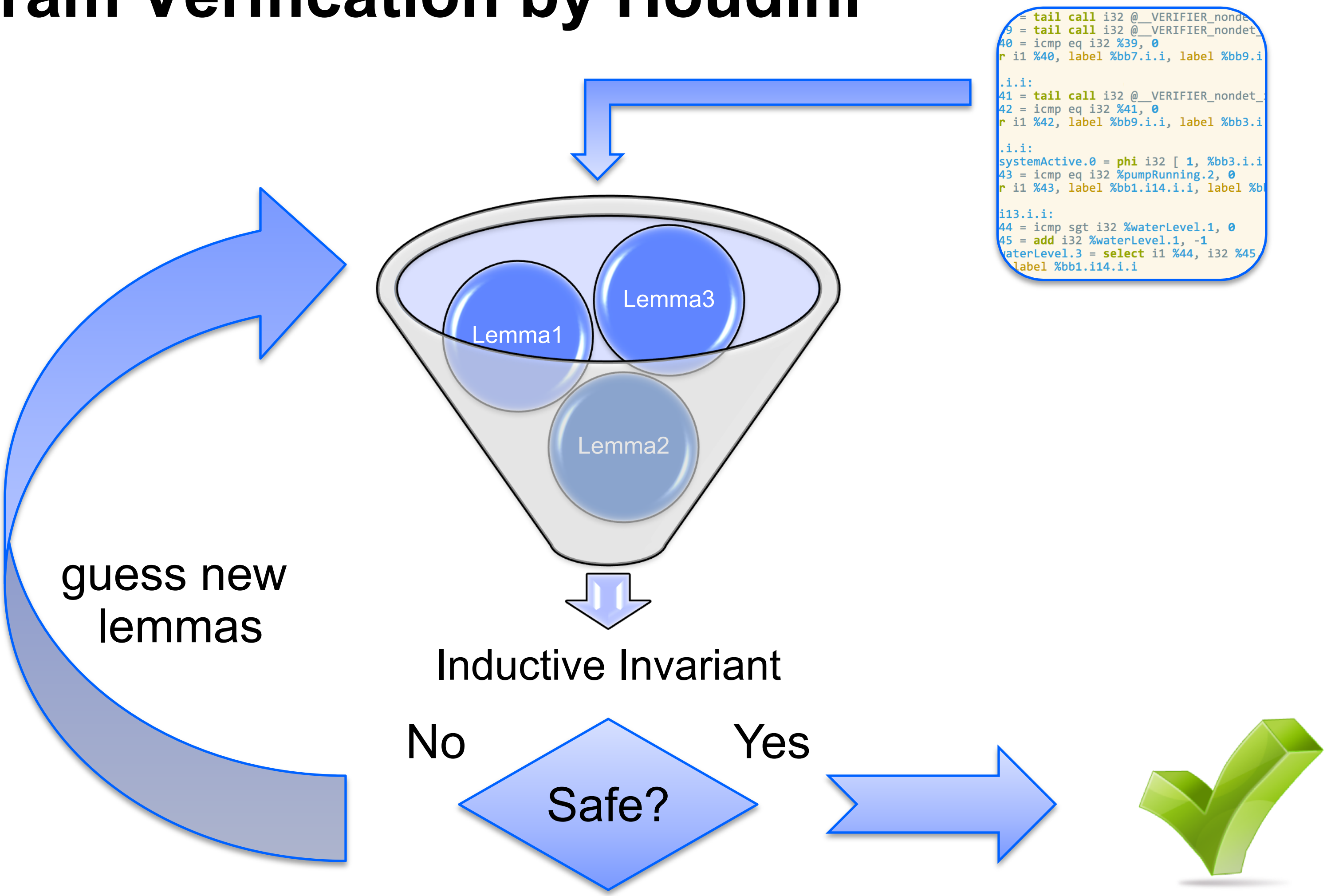
$A \Rightarrow \neg B$

$A \Rightarrow I$

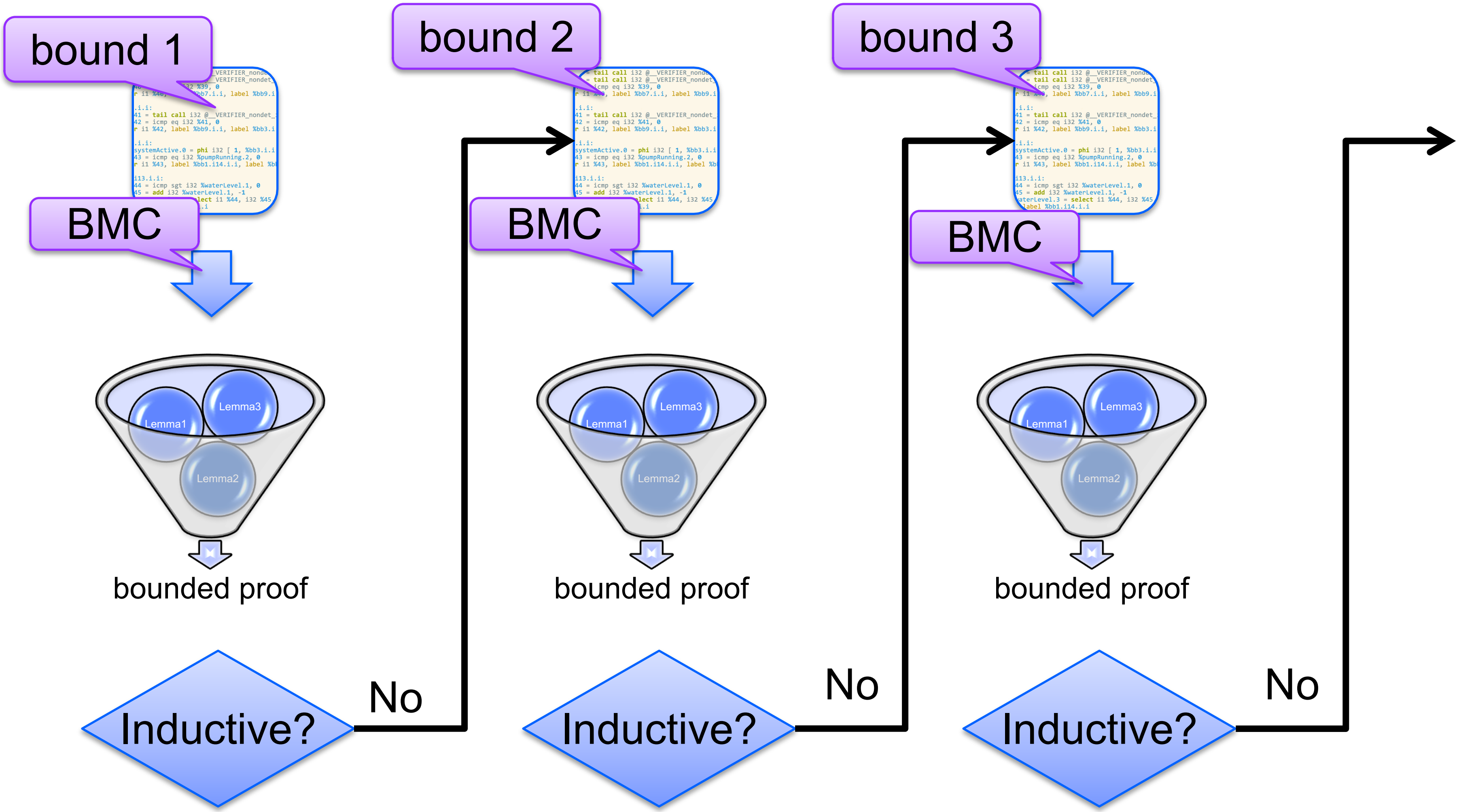
$I \Rightarrow \neg B$



Program Verification by Houdini



Verification by Successive Under-Approximation



UNIVERSITY OF WATERLOO

Interpolating Model Checking

Introduced by McMillan in 2003

- Kenneth L. McMillan: Interpolation and SAT-Based Model Checking. CAV2003: 1-13
- based on pairwise Craig interpolation

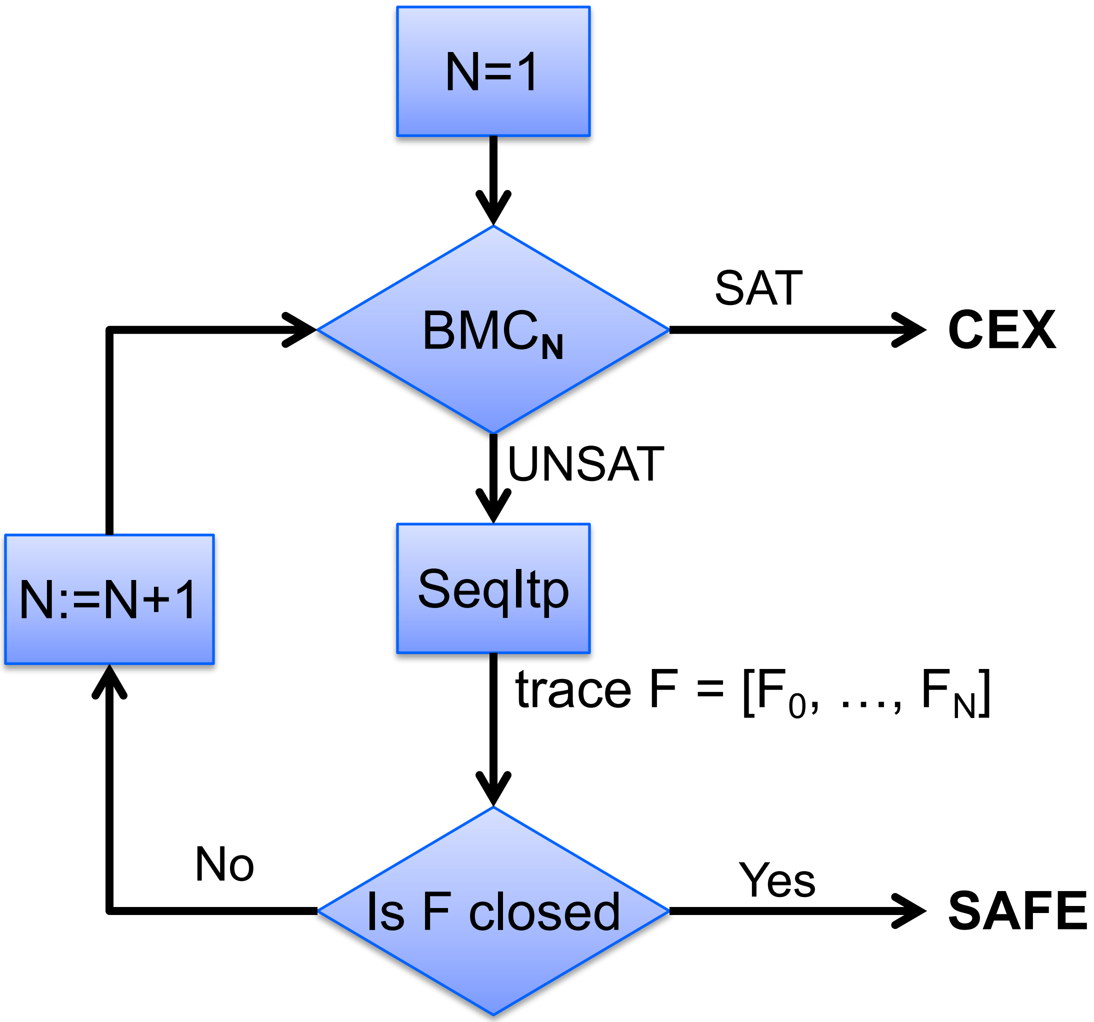
Extended to sequences and DAGs

- Yakir Vizel, Orna Grumberg: Interpolation-sequence based model checking. FMCAD 2009: 1-8
  - uses interpolation sequence
- Kenneth L. McMillan: Lazy Abstraction with Interpolants. CAV 2006: 123-136
  - IMPACT: interpolation sequence on each program path
- Aws Albarghouthi, Arie Gurfinkel, Marsha Chechik: From Under-Approximations to Over-Approximations and Back. TACAS 2012: 157-172
  - UFO: interpolation sequence on the DAG of program paths

Key Idea

- turn SAT/SMT proofs of bounded safety to inductive traces
- repeat forever until a counterexample or inductive invariant are found

IMC: Interpolating Model Checking



Inductive Trace

An *inductive trace* of a transition system $P = (V, Init, Tr, Bad)$ is a sequence of formulas $[F_0, \ldots, F_N]$ such that

- $Init \Rightarrow F_0$
- $\forall 0 \cdot i < N$, $F_i(v) \wedge Tr(v, u) \Rightarrow F_{i+1}(u)$

A trace is *safe* iff $\forall\ 0 \leq i \leq N$, $F_i \Rightarrow \neg Bad$

A trace is *monotone* iff $\forall\ 0 \cdot i < N$, $F_i \Rightarrow F_{i+1}$

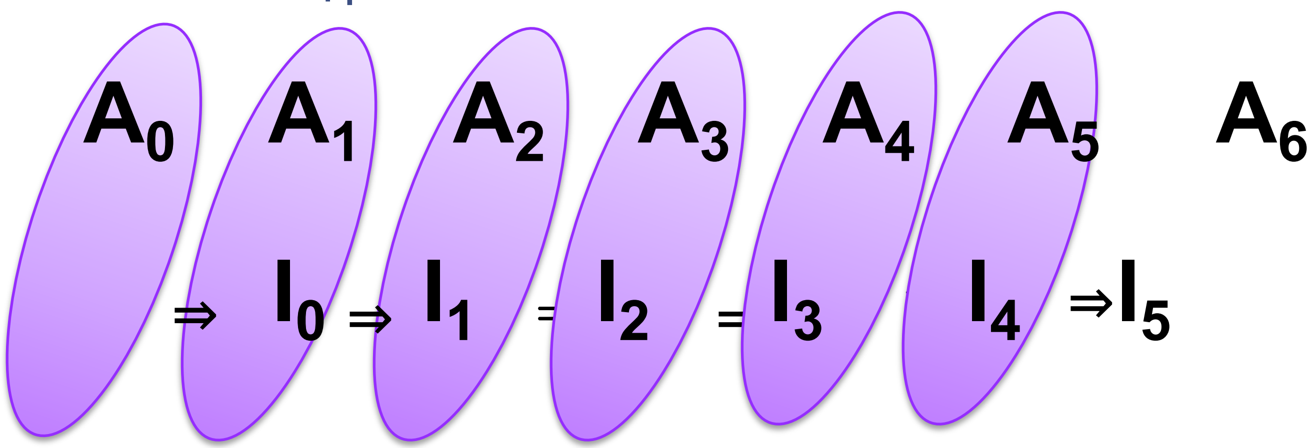
A trace is *closed* iff $\exists\ 1 \leq i \leq N$, $F_i \Rightarrow (F_0 \vee \ldots \vee F_{i-1})$

A transition system P is SAFE iff it admits a safe closed trace

Interpolation Sequence

Given a sequence of formulas $\mathbf{A = \{A_i\}_{i=0}^n}$, an ***interpolation sequence*** $\mathbf{ItpSeq(A) = \{I_1, \ldots, I_{n-1}\}}$ is a sequence of formulas such that

- $\mathbf{I_k}$ is an **ITP** $\mathbf{(A_0 \wedge \ldots \wedge A_{k-1}, \quad A_k \wedge \ldots \wedge A_n)}$, and
- $\mathbf{\forall\ k<n\ .\ I_k \wedge A_{k+1} \Rightarrow I_{k+1}}$



Can compute by pairwise interpolation applied to different cuts of a fixed resolution proof (very robust property of interpolation)

From Interpolants to Traces

A Sequence Interpolant of a BMC instance is an inductive trace

BMC$_N$

$$(\mathit{Init}(v_0))_0 \;\wedge\; (\mathit{Tr}(v_0,v_1))_1 \;\wedge \ldots \wedge\; (\mathit{Tr}(v_{N-1}, v_N))_N \;\wedge\; \mathit{Bad}(v_N)$$

$$F_0(v_0) \qquad\qquad F_1(v_1) \qquad\qquad\qquad\qquad F_N(v_N)$$

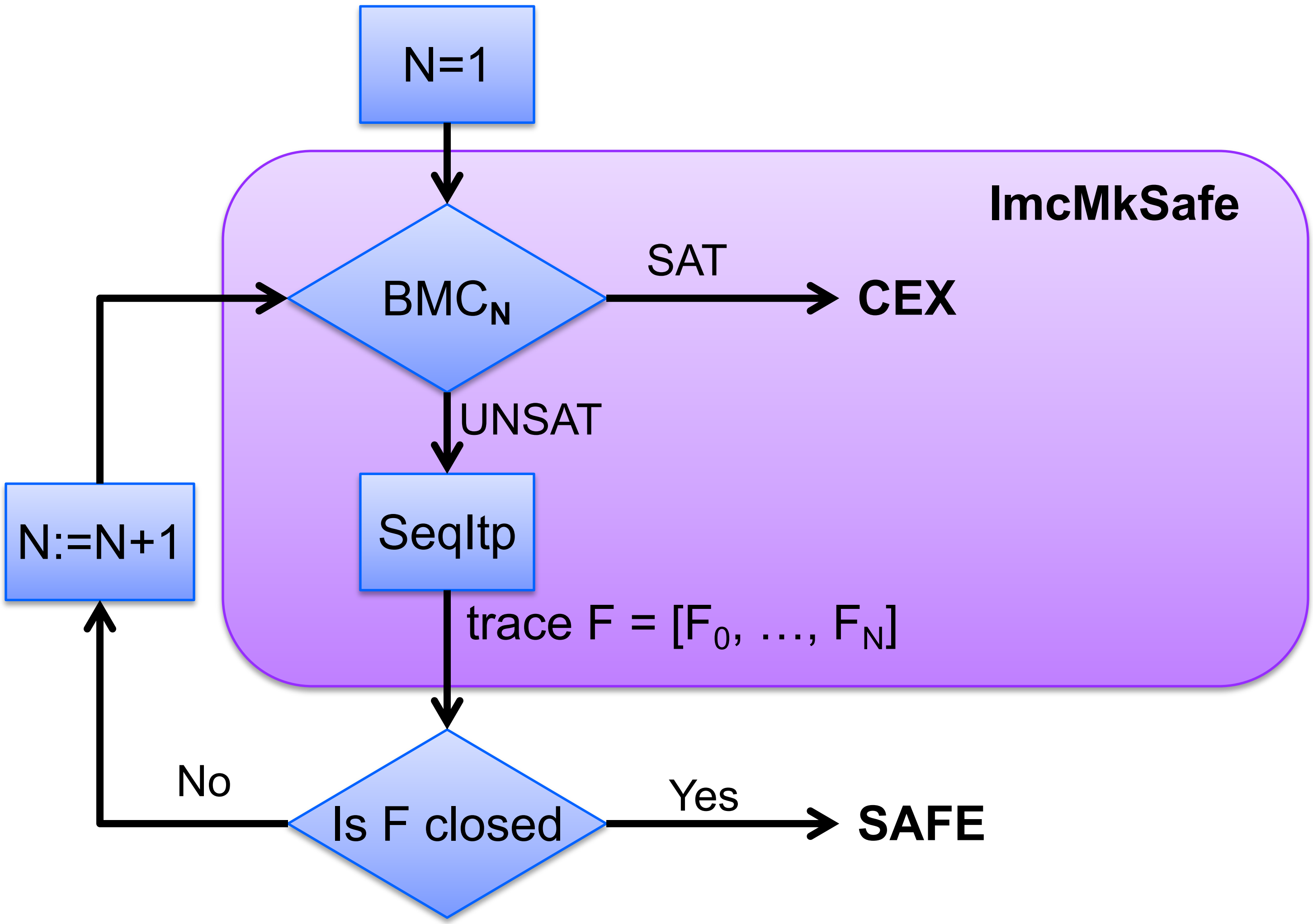
trace

A trace computed by a sequence interpolant is

- safe
- NOT necessarily monotone
- NOT necessarily closed

UNIVERSITY OF WATERLOO

IMC: Interpolating Model Checking



IMC: Strength and Weaknesses

Strength

- elegant
- global bounded safety proof
- many different interpolation algorithms available
- easy to extend to SMT theories

Weaknesses

- the naïve version does not converge easily
  - interpolants are weaker towards the end of the sequence
- not incremental
  - no information is reused between BMC queries
- size of interpolants
- hard to guide

IC3: Property Directed Reachability

IC3: A SAT-based Hardware Model Checker

- Incremental Construction of Inductive Clauses for Indubitable Correctness
- A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011

PDR: Explained and extended the implementation

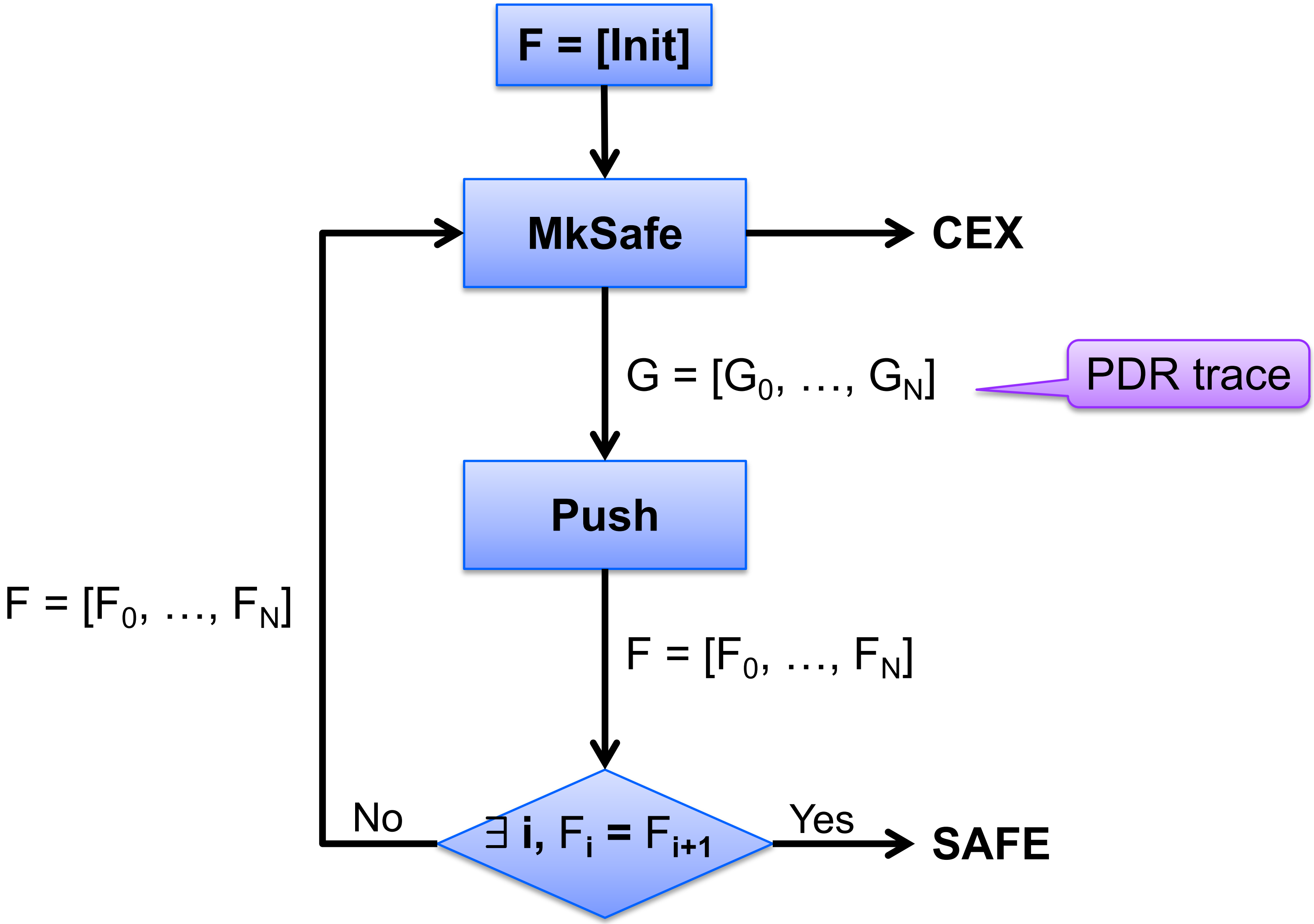
- Property Directed Reachability
- N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

Very active area of research

Key Idea:

- carefully manage SAT solving while building an inductive proof one inductive lemma at a time

IC3/PDR



F = [Init]
MkSafe
CEX
G = [G0, …, GN]
PDR trace
Push
F = [F0, …, FN]
F = [F0, …, FN]
No
∃ i, Fi = Fi+1
Yes
SAFE

IC3, PDR, and Friends (1)

IC3: A SAT-based Hardware Model Checker

- Incremental Construction of Inductive Clauses for Indubitable Correctness
- A. Bradley: SAT-Based Model Checking without Unrolling. VMCAI 2011

PDR: Explained and extended the implementation

- Property Directed Reachability
- N. Eén, A. Mishchenko, R. K. Brayton: Efficient implementation of property directed reachability. FMCAD 2011

PDR with Predicate Abstraction (easy extension of IC3/PDR to SMT)

- A. Cimatti, A. Griggio, S. Mover, St. Tonetta: IC3 Modulo Theories via Implicit Predicate Abstraction. TACAS 2014
- J. Birgmeier, A. Bradley, G. Weissenbacher: Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). CAV 2014

IC3, PDR, and Friends (2)

GPDR: Non-Linear CHC with Arithmetic constraints

- Generalized Property Directed Reachability
- K. Hoder and N. Bjørner: Generalized Property Directed Reachability. SAT 2012

SPACER: Non-Linear CHC with Arithmetic

- fixes an incompleteness issue in GPDR and extends it with under-approximate summaries
- A. Komuravelli, A. Gurfinkel, S. Chaki: SMT-Based Model Checking for Recursive Programs. CAV 2014

PolyPDR: Convex models for Linear CHC

- simulating Numeric Abstract Interpretation with PDR
- N. Bjørner and A. Gurfinkel: Property Directed Polyhedral Abstraction. VMCAI 2015

ArrayPDR: CHC with constraints over Airthmetic + Arrays

- Required to model heap manipulating programs
- A. Komuravelli, N. Bjørner, A. Gurfinkel, K. L. McMillan:Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. FMCAD 2015

IC3, PDR, and Friends (3)

Quip: Forward Reachable States + Conjectures

- Use both forward and backward reachability information
- A. Gurfinkel and A. Ivrii: Pushing to the Top. FMCAD 2015

Avy: Interpolation with IC3

- Use SAT-solver for blocking, IC3 for pushing
- Y. Vizel, A. Gurfinkel: Interpolating Property Directed Reachability. CAV 2014

uPDR: Constraints in EPR fragment of FOL

- Universally quantified inductive invariants (or their absence)
- A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, S. Shoham: Property-Directed Inference of Universal Invariants or Proving Their Absence. CAV 2015

Quic3: Universally quantified invariants for LIA + Arrays

- Extending Spacer with quantified reasoning
- A. Gurfinkel, S. Shoham, Y. Vizel: Quantifiers on Demand. ATVA 2018

IC3

(Bradley, VMCAI 2011)

IC3 = **I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness

The Goal: Find an Inductive Invariant stronger than P

- Recall: F is an inductive invariant stronger than P if
  - $INIT \Rightarrow F$
  - $F \wedge T \Rightarrow F'$
  - $F \Rightarrow P$

by learning relatively inductive facts (incrementally)

In a property directed manner

- also called "Property Directed Reachability" (**PDR**)

PDR Trace

Recall that an *inductive trace* of a transition system P = (V, Init, Tr, Bad) is a sequence of formulas $[F_0, \ldots, F_N]$ such that

- $Init \Rightarrow F_0$
- $\forall\ 0 \leq i < N\ ,\ F_i(v) \wedge Tr\ (v, u) \Rightarrow F_{i+1}\ (u)$

A trace is *clausal* if every frame F_i is in CNF

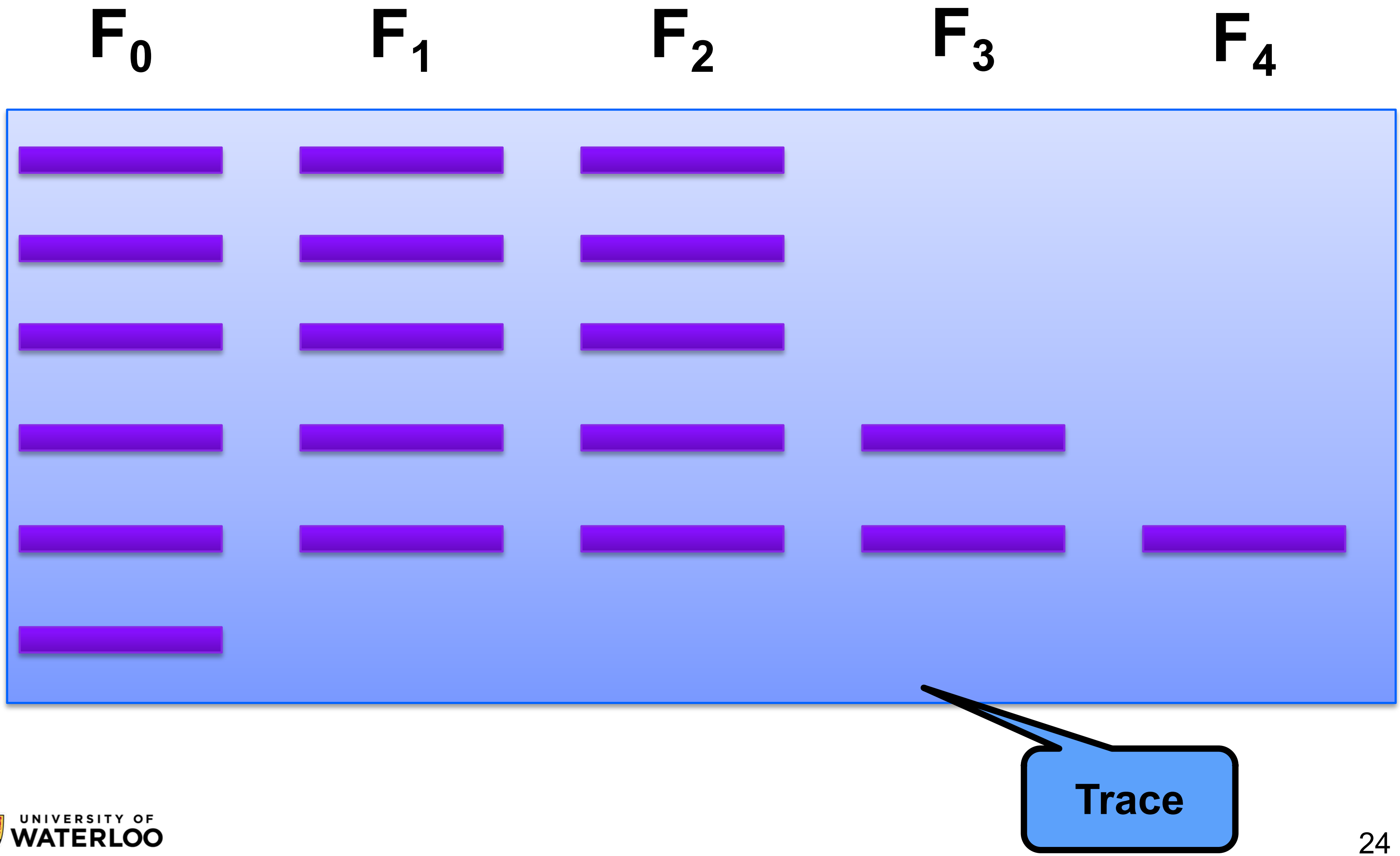
A delta-compressed trace (or *δ-trace*) is a sequence of clauses s.t.

- each clause c belongs to a unique frame F_i
- $\forall\ 0 \leq i \leq n\ ,\ \forall\ j < i\ ,\ (c \in F_i) \Rightarrow (c \notin F_j)$

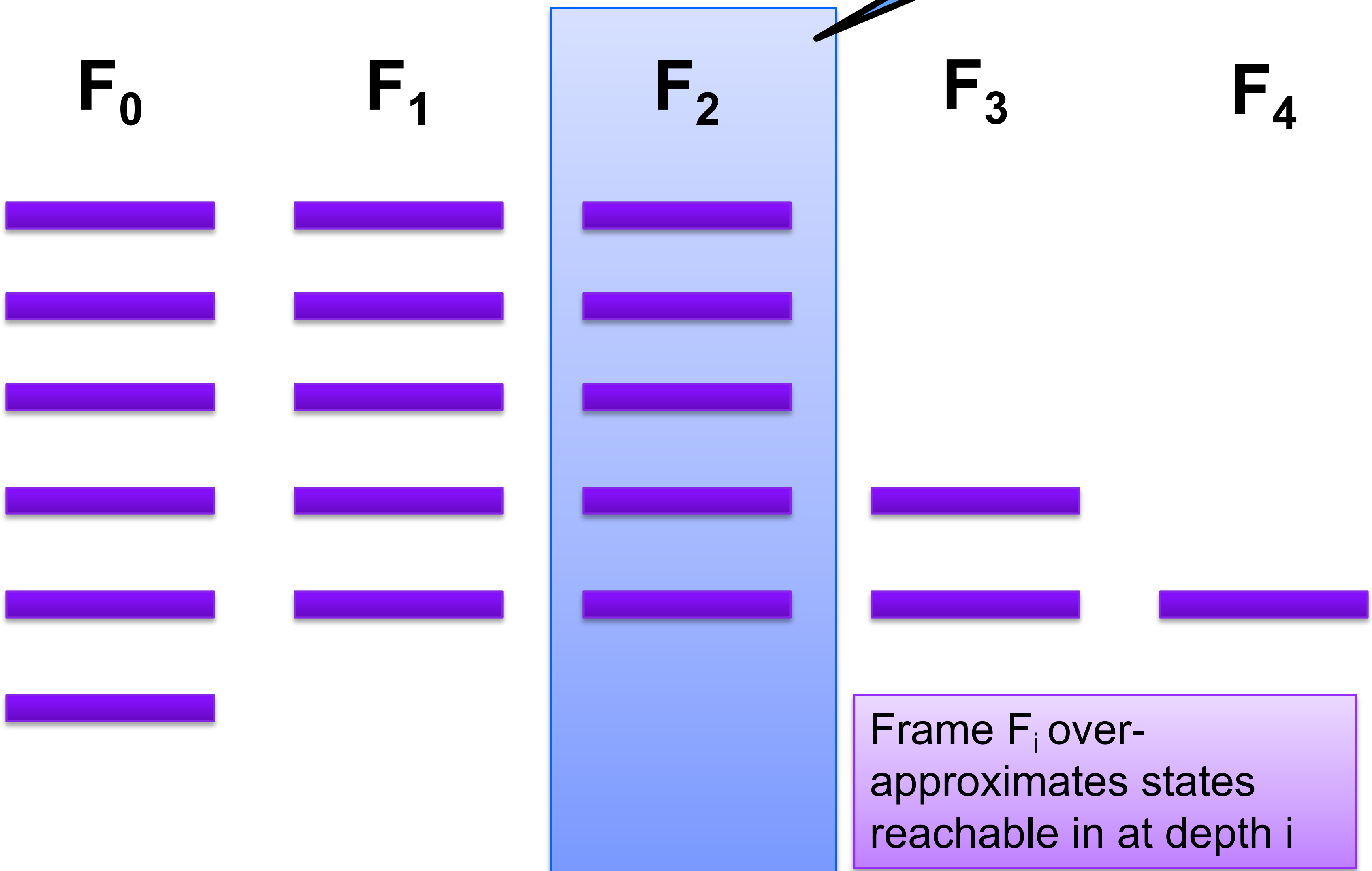
A PDR trace is a monotone, clausal, safe (up to N-1)

- PDR trace is often represented compactly by a δ-trace

PDR Trace Pictionary

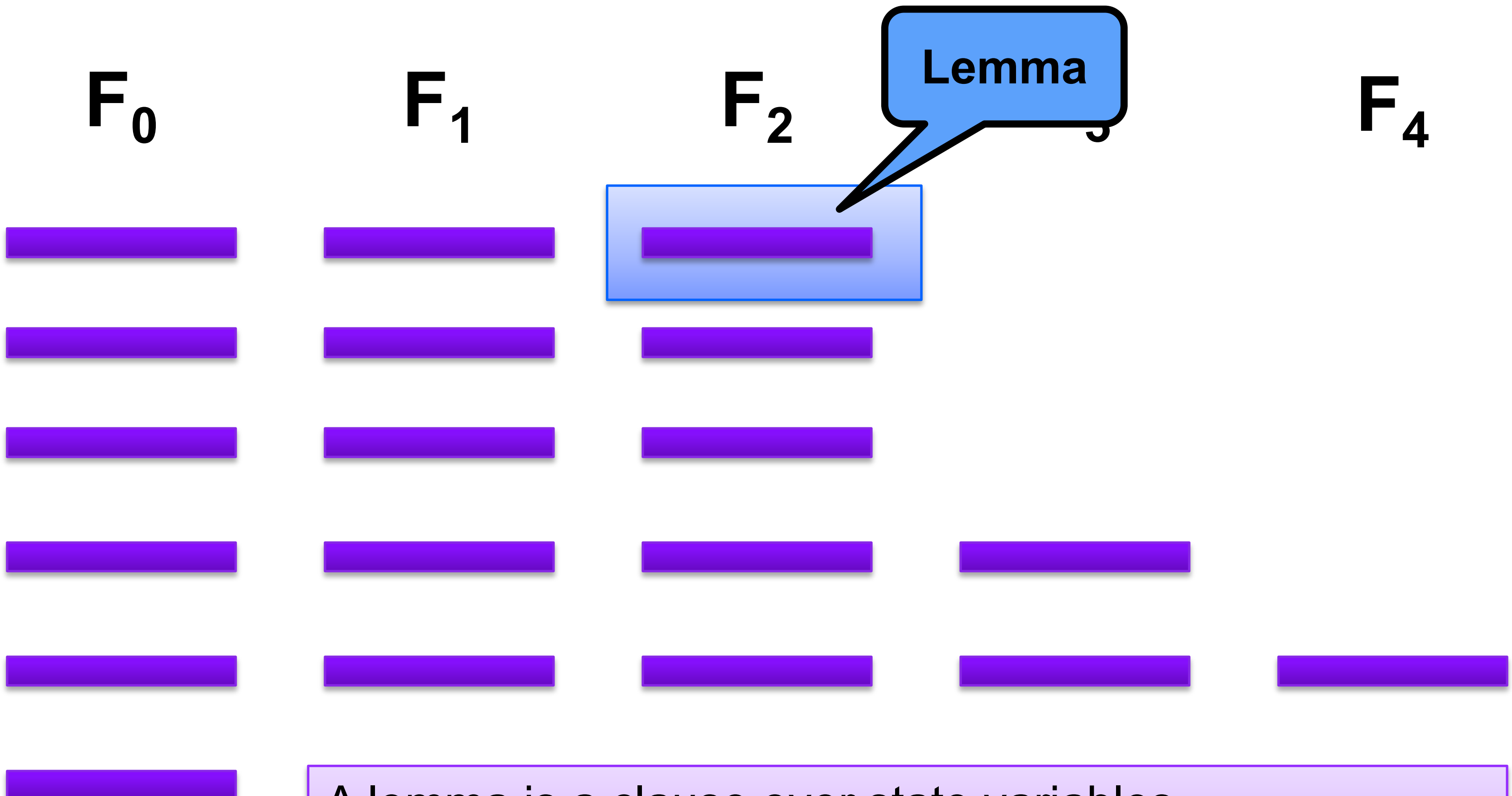


PDR Trace Pictionary: Frame



Frame F_i over-approximates states reachable in at depth i

PDR Trace Pictionary: Lemma

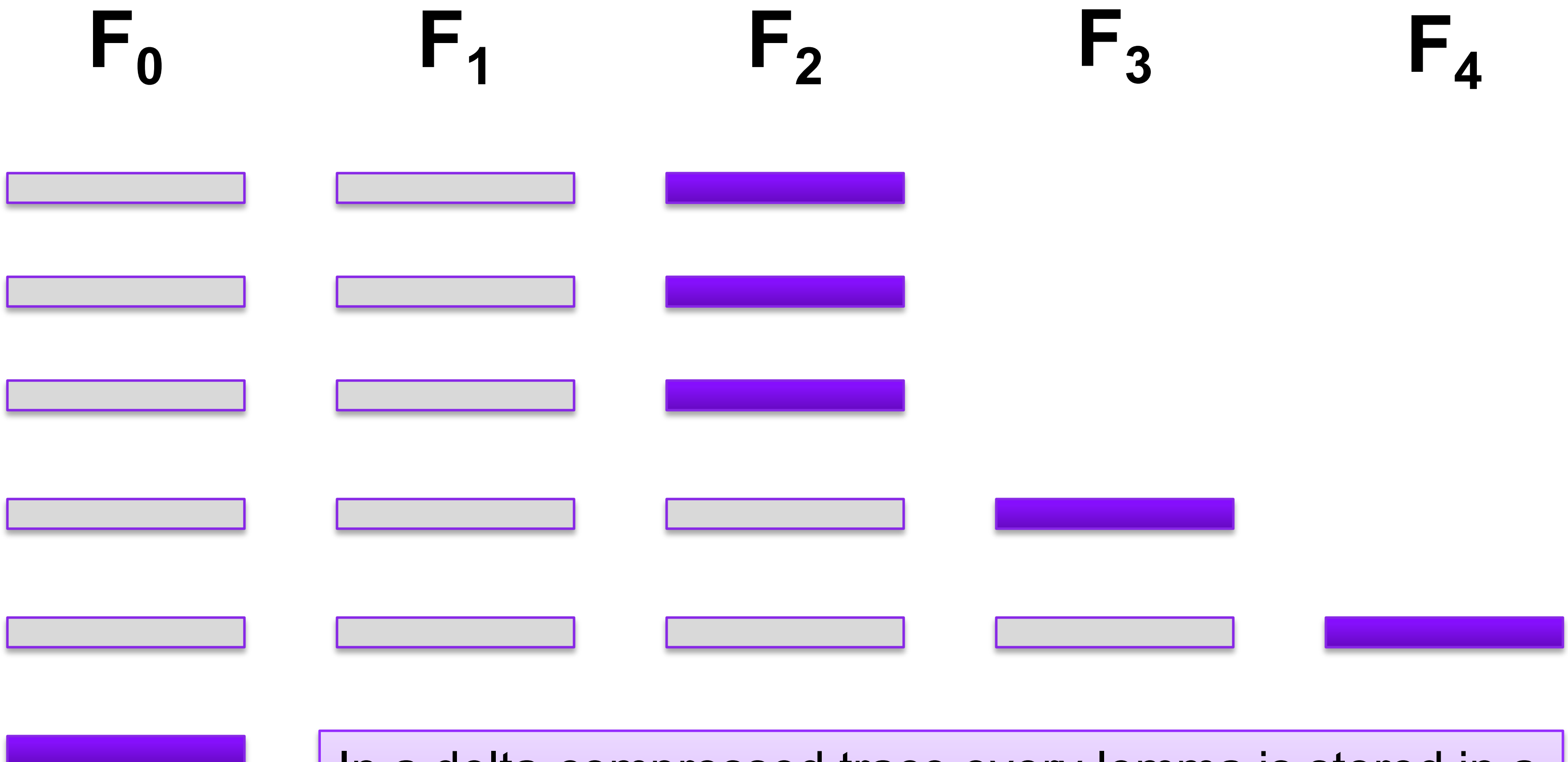


A lemma is a clause over state variables

A lemma blocks (or excludes) bad states

A trace is monotone if lemmas are shared in frames

PDR Trace Pictionary: Delta Compression

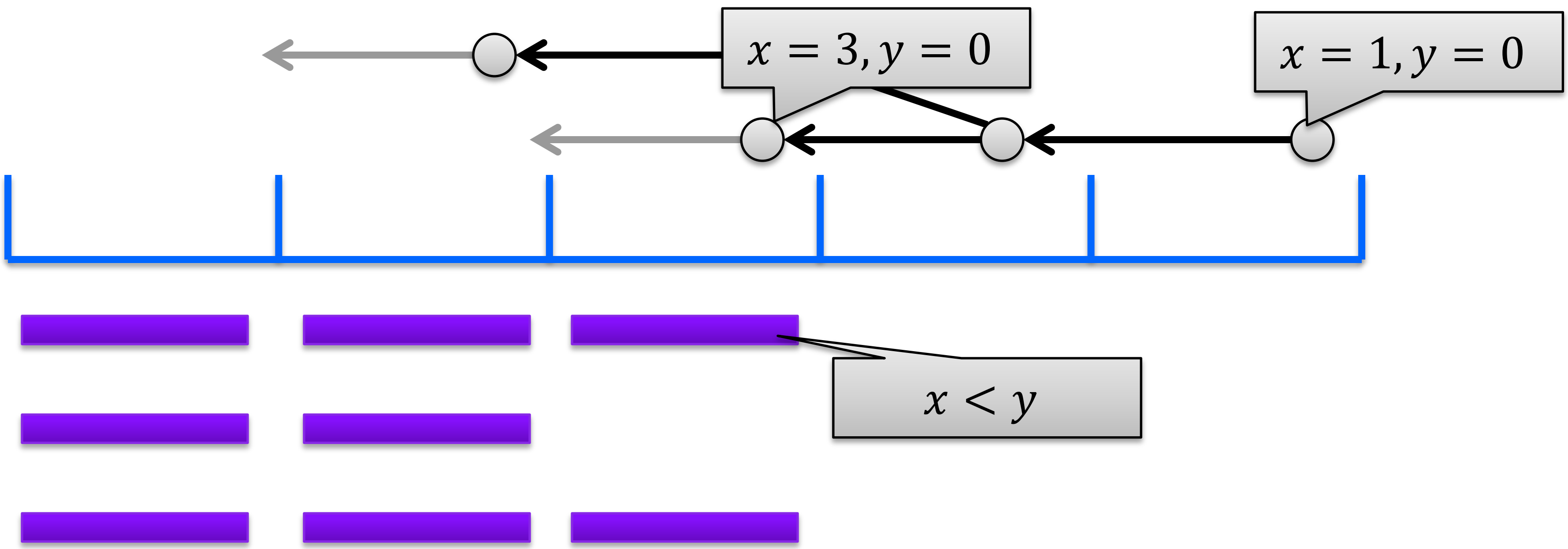


In a delta-compressed trace every lemma is stored in a frame with the largest index that it appears

A delta trace is closed (inductive) if it has an empty frame

IC3/PDR In Pictures: MkSafe

MkSafe



Predecessor

$$\text{find } M \text{ s.t. } M \models F_i \wedge Tr \wedge m'$$

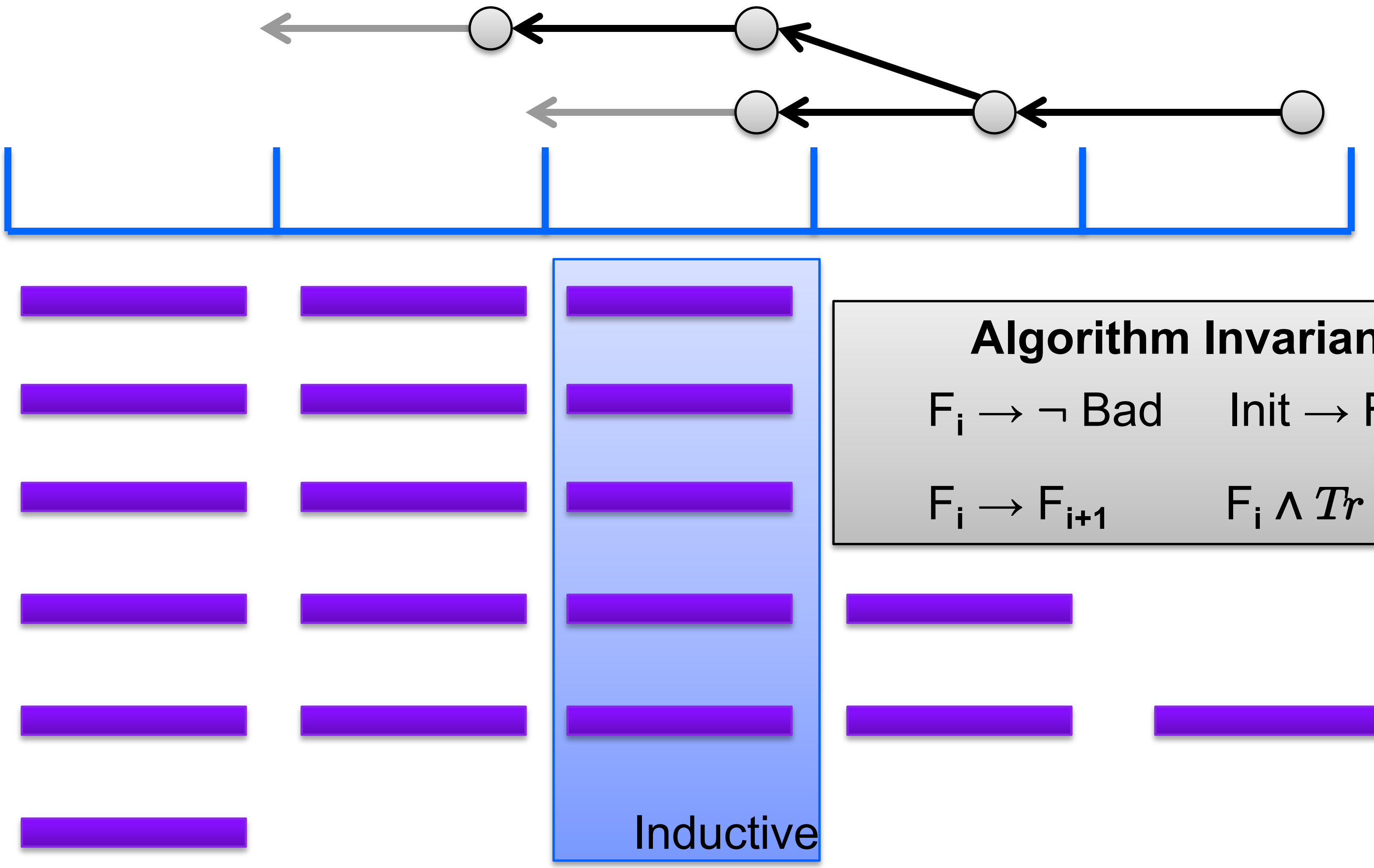
$$\text{find } m \text{ s.t. } (M \models m) \wedge (m \implies \exists V' \cdot Tr \wedge m')$$

NewLemma

$$\text{find } \ell \text{ s.t. } (F_i \wedge Tr \implies \ell') \wedge (\ell \implies \neg m)$$

IC3/PDR in Pictures: Push

Push



Algorithm Invariants

$F_i \rightarrow \neg \text{Bad}$ $\quad$ $\text{Init} \rightarrow F_i$

$F_i \rightarrow F_{i+1}$ $\quad$ $F_i \wedge \mathit{Tr} \rightarrow F_{i+1}$

$$\text{SMT-query: } \vdash \ell \wedge F_i \wedge Tr \implies \ell'$$

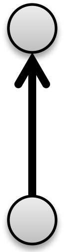
IC3 Data-Structures

A trace $F = F_0, \ldots, F_N$ is a sequence of frames.

- A frame F_i is a set of clauses. Elements of F_i are called lemmas.
- Invariants:
  - Bounded Safety: $\forall\, i < N \,.\, F_i \rightarrow \neg Bad$
  - Monotonicity: $\forall\, i < N \,.\, F_{i+1} \subseteq F_i$
  - Inductiveness: $\forall\, i < N \,.\, F_i \wedge Tr \rightarrow F'_{i+1}$

A priority queue Q of counterexamples to induction (CTI) or proof obligations (POB)

- $(m, i) \in Q$ is a pair, where m is a cube and i a level
- if $(m, i) \in Q$ then there exists a path of length (N-i) from a state in m to a state in Bad
- Q is ordered by level
  - $(m, i) < (k, j)$ iff $i < j$



UNIVERSITY OF WATERLOO

Recursive Blocking Stage in IC3

```
// Find a counterexample, or strengthen the inductive trace
// s.t. F_N ⇒ ¬s holds
IC3_recBlockCube(s, N)
    Add(Q, (s, N))
    while ¬Empty(Q) do
        (s, k) ← Pop(Q)
        if (k = 0) return "Counterexample"
        if (F_k ⇒ ¬s) continue
        if (F_k-1 ∧ Tr ∧ s') is SAT
            t ← generalized predecessor of s
            Add(Q, (t, k-1))
            Add(Q, (s, k))
        else
            ¬t ← generalize ¬s by inductive generalization (to
                                                           level m≥k)
            add ¬t to F_m
            if (m<N) Add(Q, (s, m+1))
```

Pushing stage in IC3

```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c')
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return "Proof" // F_k is a safe inductive invariant
```

PDR Strength and Weaknesses

Strengths

- elegant
- incremental
- many opportunities for guidance
  - fine-grained proof management
  - fine-grained generalization of lemmas

Weaknesses

- local backward search for a counterexample
- CNF explosion

IC3/PDR: Solving Linear (Propositional) CHC

Unreachable and Reachable

- terminate the algorithm when a solution is found

Unfold

- increase search bound by 1

Candidate

- choose a bad state in the last frame

Decide

- extend a cex (backward) consistent with the current frame
- choose an assignment ***s*** s.t. $(s \wedge F_i \wedge Tr \wedge cex')$ is SAT

Conflict

- construct a lemma to explain why cex cannot be extended
- Find a clause ***L*** s.t. $L \Rightarrow \neg cex$, $Init \Rightarrow L$, and $L \wedge F_i \wedge Tr \Rightarrow L'$

Induction

- propagate a lemma as far into the future as possible
- (optionally) strengthen by dropping literals

Termination and Progress

Unreachable If there is an $i < N$ s.t. $F_i \subseteq F_{i+1}$

return *Unreachable*.

Reachable If there is an m s.t. $\langle m, 0 \rangle \in Q$

return *Reachable*.

Unfold If $F_N \to \neg Bad$, then set $N \leftarrow N + 1$.

Candidate If for some m, $m \to F_N \wedge Bad$,

then add $\langle m, N \rangle$ to Q.

UNIVERSITY OF WATERLOO

Inductive Generalization

Conflict For $0 \le i < N$: given a candidate model $\langle m, i+1 \rangle \in Q$ and clause φ, such that $\varphi \rightarrow \neg m$, if $\mathit{Init} \rightarrow \varphi$, and $\varphi \wedge F_i \wedge \mathit{Tr} \rightarrow \varphi'$, then add φ to F_j, for $j \le i+1$.

A clause φ is inductive relative to F iff

- Init $\rightarrow \varphi$ (Initialization) and $\varphi \wedge F \wedge Tr \rightarrow \varphi'$ (Inductiveness)

Implemented by first letting $\varphi = \neg m$ and generalizing φ by iteratively dropping literals while checking the inductiveness condition

Theorem: Let $F_0, F_1, \ldots, F_N$ be a valid IC3 trace. If φ is inductive relative to F_i, $0 \cdot i < N$, then, for all $j \cdot i$, φ is inductive relative to F_j.

- Follows from the monotonicity of the trace
  - if $j < i$ then $F_j \rightarrow F_i$
  - if $F_j \rightarrow F_i$ then $(\varphi \wedge F_i \wedge Tr \rightarrow \varphi') \rightarrow (\varphi \wedge F_j \wedge Tr \rightarrow \varphi')$

Prime Implicants

A formula φ is an *implicant* of a formula ψ iff $\varphi \Rightarrow \psi$

A *propositional implicant* of ψ is a conjunction of literals φ such that φ is an implicant of ψ

- φ is a conjunction of literals
- $\varphi \Rightarrow \psi$
- φ is a partial assignment that makes ψ true

A propositonal implicant φ of ψ is called *prime* if no subset of φ is an implicant of ψ

- φ is a conjunction of literals
- $\varphi \Rightarrow \psi$
- $\forall$ p . (p $\neq \varphi \land \varphi \Rightarrow$ p) $\Rightarrow$ (p $\not\Rightarrow \psi$)

Generalizing Predecessors

> **Decide** If $\langle m, i+1 \rangle \in Q$ and there are m_0 and m_1 s.t. $m_1 \rightarrow m$, $m_0 \wedge m_1'$ is satisfiable, and $m_0 \wedge m_1' \rightarrow F_i \wedge Tr \wedge m'$, then add $\langle m_0, i \rangle$ to Q.

Decide rule chooses a (generalized) predecessor m_0 of m that is consistent with the current frame

Simplest implementation is to extract a predecessor m_o from a satisfying assignment of $M \models F_i \wedge Tr \wedge m'$

- m_0 cab be further generalized using ternary simulation by dropping literals and checking that m' remains forced

An alternative is to let m_0 be an implicant (not necessarily prime) of $F_i \wedge \exists X'.(Tr \wedge m')$

- finding a prime implicant is difficult because of the existential quantification
- we settle for an arbitrary implicant. The side conditions ensure it is not trivial

Strengthening a trace

Induction For $0 \leq i < N$ and a clause $(\varphi \vee \psi) \in F_i$, if $\varphi \notin F_{i+1}$, $\mathit{Init} \to \varphi$ and $\varphi \wedge F_i \wedge \mathit{Tr} \to \varphi'$, then add φ to F_j, for each $j \leq i+1$.

Also known as **Push** or **Propagate**

Bounded safety proofs are usually very weak towards the end

- not much is needed to show that error will not happen in one or two steps

This tends to make them non-inductive

- a weakness of interpolation-based model checking, like IMPACT
- in IMPACT, this is addressed by forced covering heuristic

Induction "applies" forced cover one lemma at a time

- whenever all lemmas are pushed F_{i+1} is inductive (and safe)
- (optionally) combine strengthening with generalization

Implementation

- Apply Induction from 0 to N whenever **Conflict** and **Decide** are not applicable

UNIVERSITY OF WATERLOO

Long Counterexamples

Leaf If $\langle m, i\rangle \in Q$, $0 < i < N$ and $F_{i-1} \wedge Tr \wedge m'$ is unsatisfiable, then add $\langle m, i+1\rangle$ to Q.

Also known as **ReQueue**

Whenever a counterexample *m* is blocked at level *i*, it is known that

- there is no path of length *i* from *Init* to *m* (because got blocked)
- there is a path of length *(N-i)* from *m* to *Bad*

Can check whether there exists a path of length *(i+1)* from *Init* to *m*

- (**Leaf)** check eagerly by placing the CTI back into the queue at a higher level
- **(No Leaf)** check lazily by waiting until the same (or similar) CTI is discovered after N is increased by **Unfold**

Leaf allows IC3 to discover counterexamples much longer than the current unfolding depth N

- each CTI re-enqueued by **Leaf** adds one to the depth of the longest possible counterexample found
- a real counterexample might chain through multiple such CTI's

UNIVERSITY OF WATERLOO

Queue Management for Long Counterexamples

A queue element is a triple *(m, i, d)*

- *m* is a CTI, *i* a level, *d* a depth

Decide sets *m* and *i* as before, and sets *d* to 0

Leaf increases *i* and *d* by one

- *i* determines how far the CTI can be pushed back
- *d* counts number of times the CTI was pushed forward

Queue is ordered first by level, then by depth

- $(m, i, d) < (k, j, e) \Leftrightarrow i < j$ ç $(i=j \land d < e)$

Overall exploration mimics iterative deepening with non-uniform exploration depth

- go deeper each time before backtracking

UNIVERSITY OF WATERLOO

Recursive Blocking Stage in IC3

```
// Find a counterexample, or strengthen the inductive trace
// s.t. F_N ⇒ ¬s holds
IC3_recBlockCube(s, N)
    Add(Q, (s, N))
    while ¬Empty(Q) do
        (s, k) ← Pop(Q)
        if (k = 0) return "Counterexample"
        if (F_k ⇒ ¬s) continue
        if (F_k-1 ∧ Tr ∧ s') is SAT
            t ← generalized predecessor of s
            Add(Q, (t, k-1))
            Add(Q, (s, k))
        else
            ¬t ← generalize ¬s by inductive generalization (to
                                                         level m≥k)
            add ¬t to F_m
            if (m<N) Add(Q, (s, m+1))
```

Pushing stage in IC3

```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c')
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return "Proof" // F_k is a safe inductive invariant
```

Public IC3 Implementations

Spacer engine in Z3 (Arie)

- https://github.com/Z3Prover/z3/tree/master/src/muz/spacer
- theories and constrained horn clauses

IC3Ref (A. Bradley)

- https://github.com/arbrad/IC3ref
- IC3 reference implementation

PDR in Abc (A. Mishchenko)

- https://github.com/berkeley-abc/abc/tree/master/src/proof/pdr
- PDR implementation

IC3IA (A. Griggio)

- https://es-static.fbk.eu/people/griggio/ic3ia/index.html
- IC3 with Implicit Predicate Abstraction

Tip (N. Sörensson)

- https://github.com/niklasso/tip

State-based presentation of IC3

IC3: AGAIN

IC3 Basics

Iteratively compute Over-Approximated Reachability Sequence (OARS) $<F_0, F_1, \ldots, F_{k+1}>$ s.t.

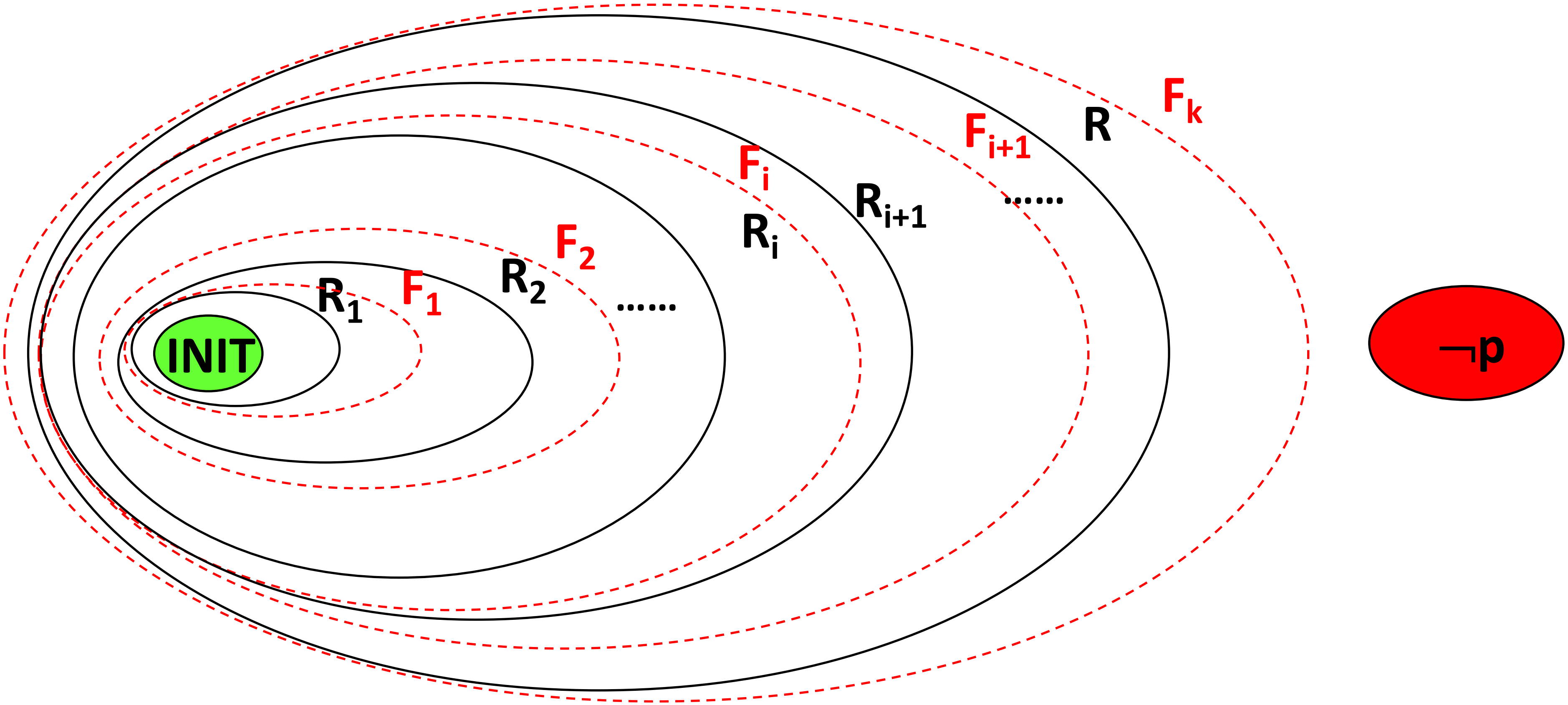
- F_0 = INIT
- $F_i \Rightarrow F_{i+1}$ — monotone: $F_i \subseteq F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$ — inductive: simulates one forward step
- $F_i \Rightarrow P$ — safe: p is an invariant up to k+1

F_i - CNF formula given as a set of clauses

F_i over-approximates R_i

- If $F_{i+1} \Rightarrow F_i$ then fixpoint: F_i is an inductive invariant

OARS (aka Inductive Trace)



$F_{i+1}(V') \Leftarrow F_i\,(V) \wedge T(V,V')$

If $F_{k+1} \equiv F_k$ then F_k is an inductive invariant

UNIVERSITY OF WATERLOO

IC3 Basics (cont.)

c is inductive relative to F if

- INIT $\Rightarrow$ c
- F $\wedge$ c $\wedge$ T $\Rightarrow$ c'

Notation:

- cube s: conjunction of literals
  - $v_1 \wedge v_2 \wedge \neg v_3$ - Represents a state
- s is a cube => **¬s is a clause** (DeMorgan)

UNIVERSITY OF WATERLOO

IC3 - Initialization

OARS:

- $F_0 = INIT$
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

Check satisfiability of the two formulas:

- $INIT \wedge \neg P$
- $INIT \wedge T \wedge \neg P'$

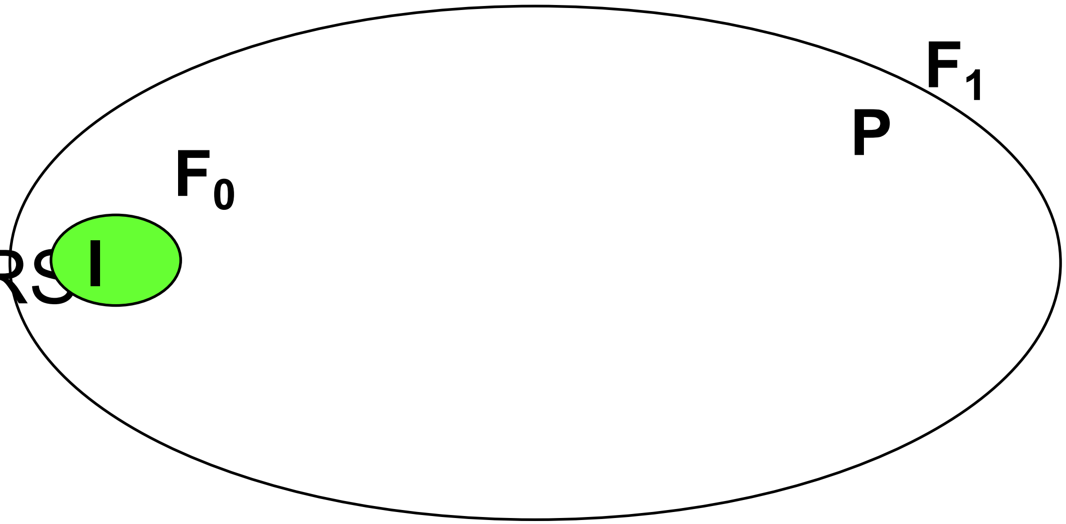
If at least one is satisfiable: cex found

If both are unsatisfiable then:

- $INIT \Rightarrow P$
- $INIT \wedge T \Rightarrow P'$

Therefore

- $F_0 = INIT$, $F_1 = P$
  - $\langle F_0, F_1 \rangle$ is an OARS



IC3 - Iteration

OARS:

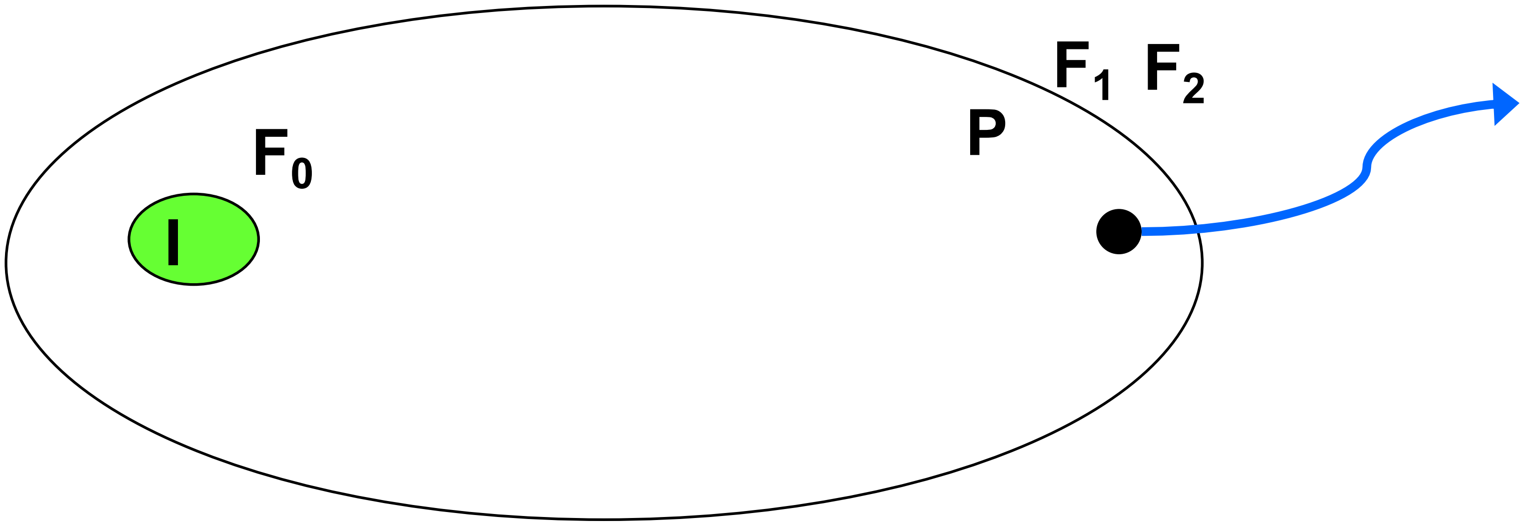
- $F_0 = INIT$
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

Our OARS contains F_0 and F_1

Initialize F_2 to P

- If P is an inductive invariant – done! ☺
- Otherwise: $F_1 \wedge T \not\Rightarrow F'_2$

  => F_1 should be strengthened



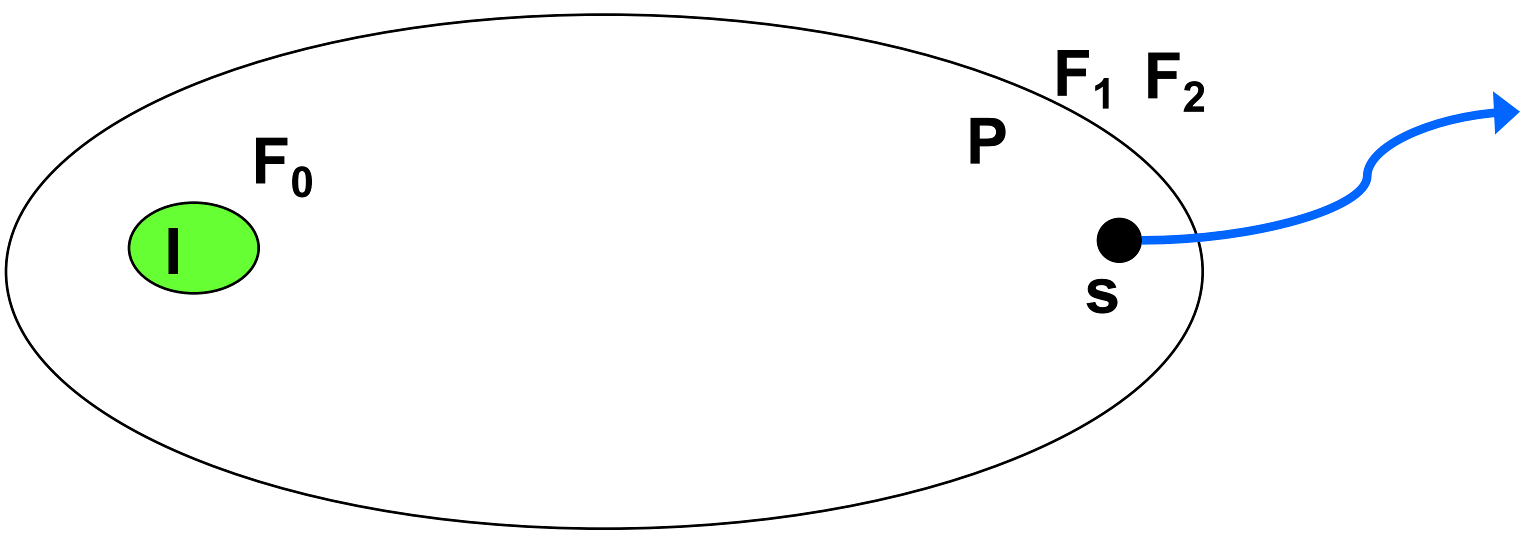
IC3 - Iteration

OARS:

- $F_0 = INIT$
- $F_i \Rightarrow F_{i+1}$
- $F_i \wedge T \Rightarrow F'_{i+1}$
- $F_i \Rightarrow P$

If P is not an inductive invariant

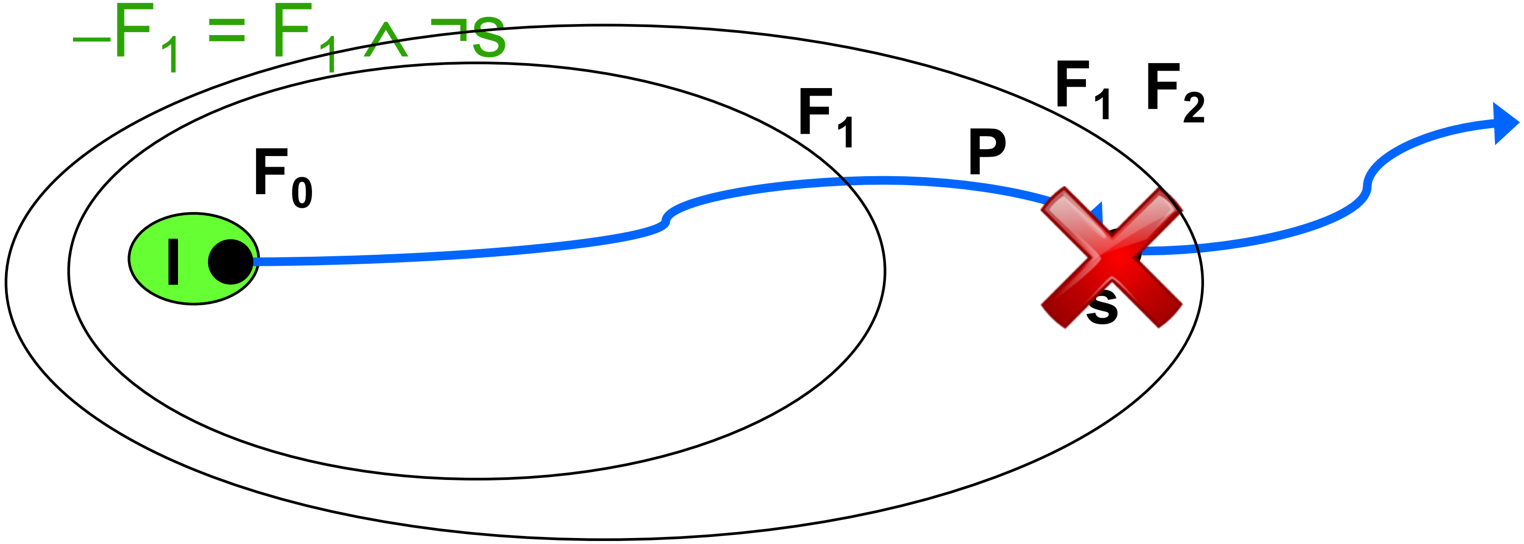
- $F_1 \wedge T \wedge \neg P'$ is satisfiable
  - $(F \wedge T \wedge \neg P')$ sat IFF $(F \wedge T \Rightarrow P')$ not valid
- From the satisfying assignment get a state s that can reach a bad state



IC3 - Iteration

Is s reachable in one transition from the previous set?

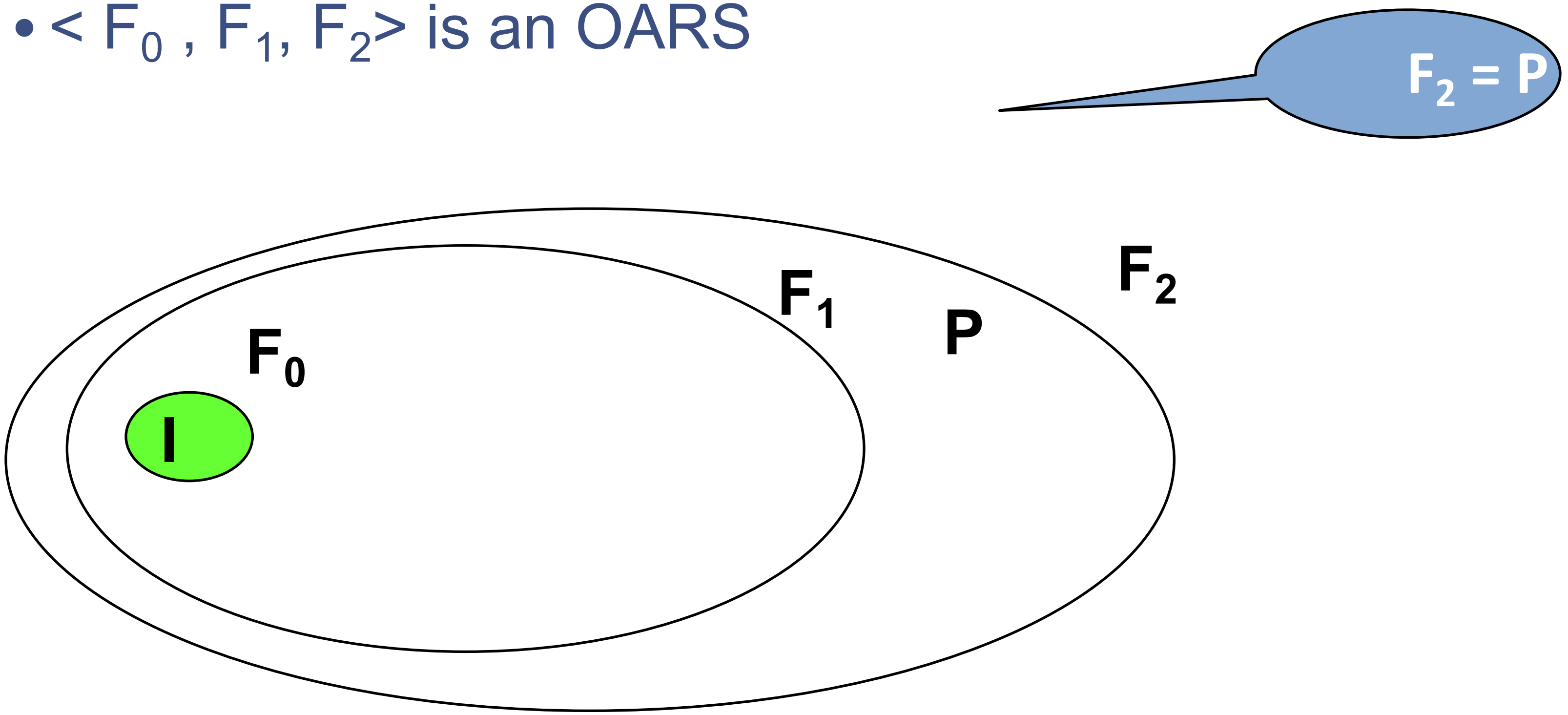
- backward search: Check $F_0 \wedge T \wedge s'$
- If satisfiable, s is reachable from F_0 : CEX
- Otherwise, block s, i.e. remove it from F_1
  - $F_1 = F_1 \wedge \neg s$



IC3 - Iteration

Iterate this process until $F_1 \wedge T \wedge \neg P'$ becomes unsatisfiable

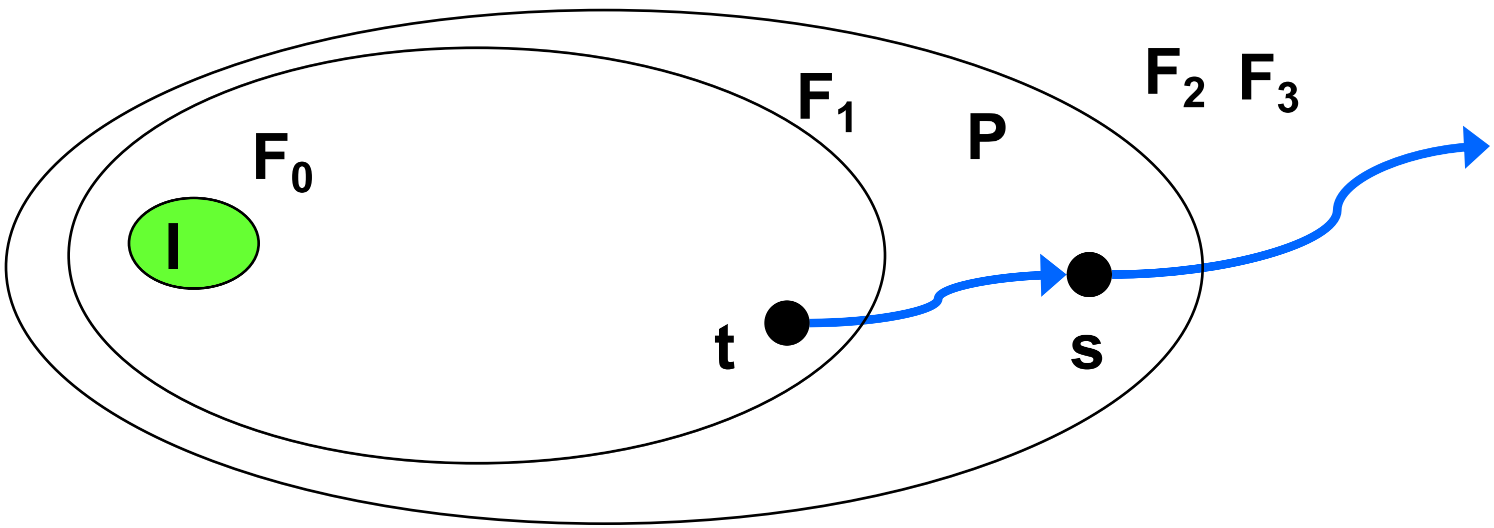
- $F_1 \wedge T \Rightarrow P'$ holds
- $< F_0 , F_1, F_2>$ is an OARS



IC3 - Iteration

New iteration, initialize F_3 to P, check $F_2 \wedge T \wedge \neg P'$

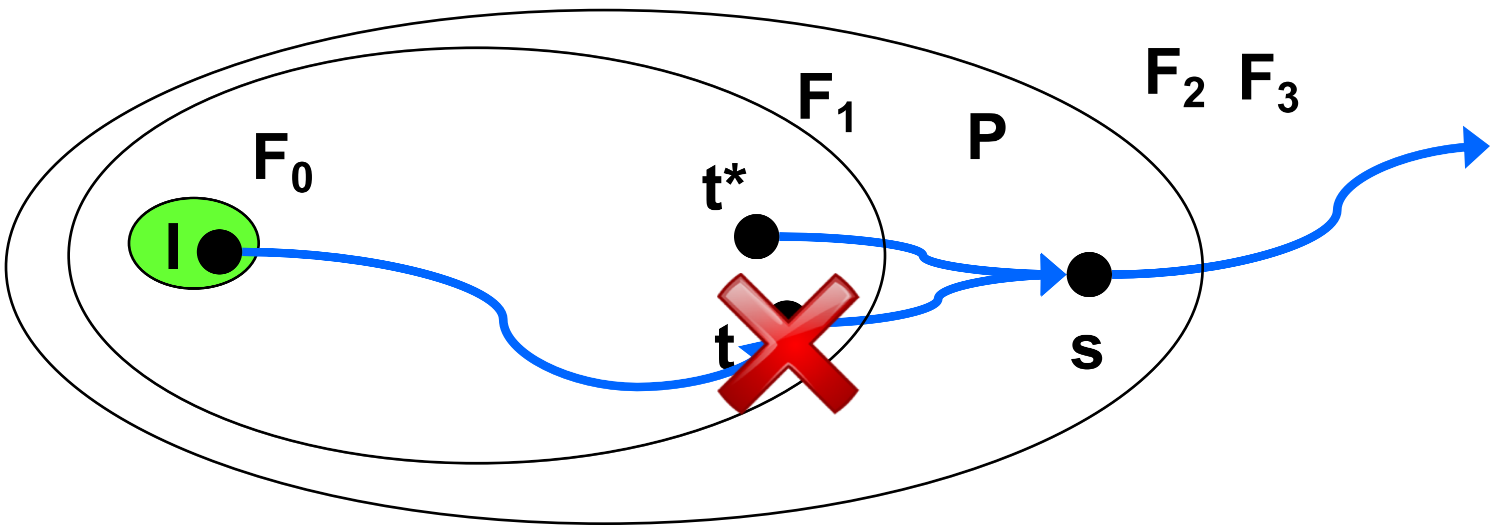
- If satisfiable, get s that can reach ¬P
- Now check if s can be reached from F_1 by $F_1 \wedge T \wedge s'$
- **If it can be reached, get t and try to block it**



IC3 - Iteration

To block t, check $F_0 \wedge T \wedge t'$

- If satisfiable, a CEX
- If not, t is blocked, get a “new” t* by $F1 \wedge T \wedge s'$ and try to block t*

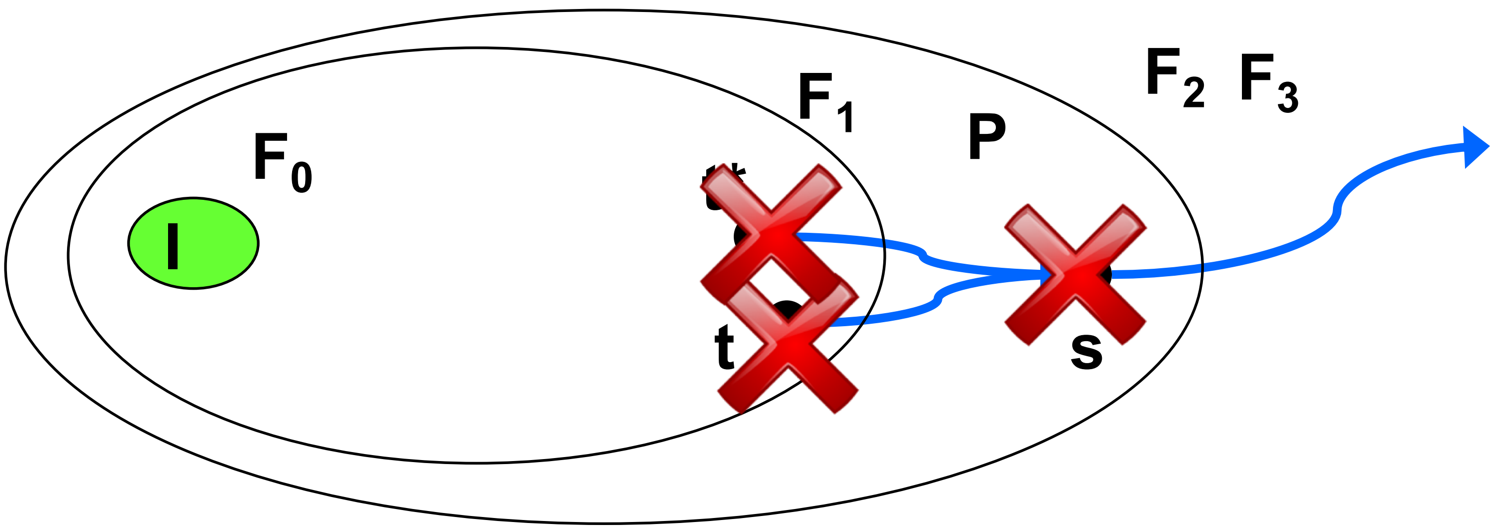


IC3 - Iteration

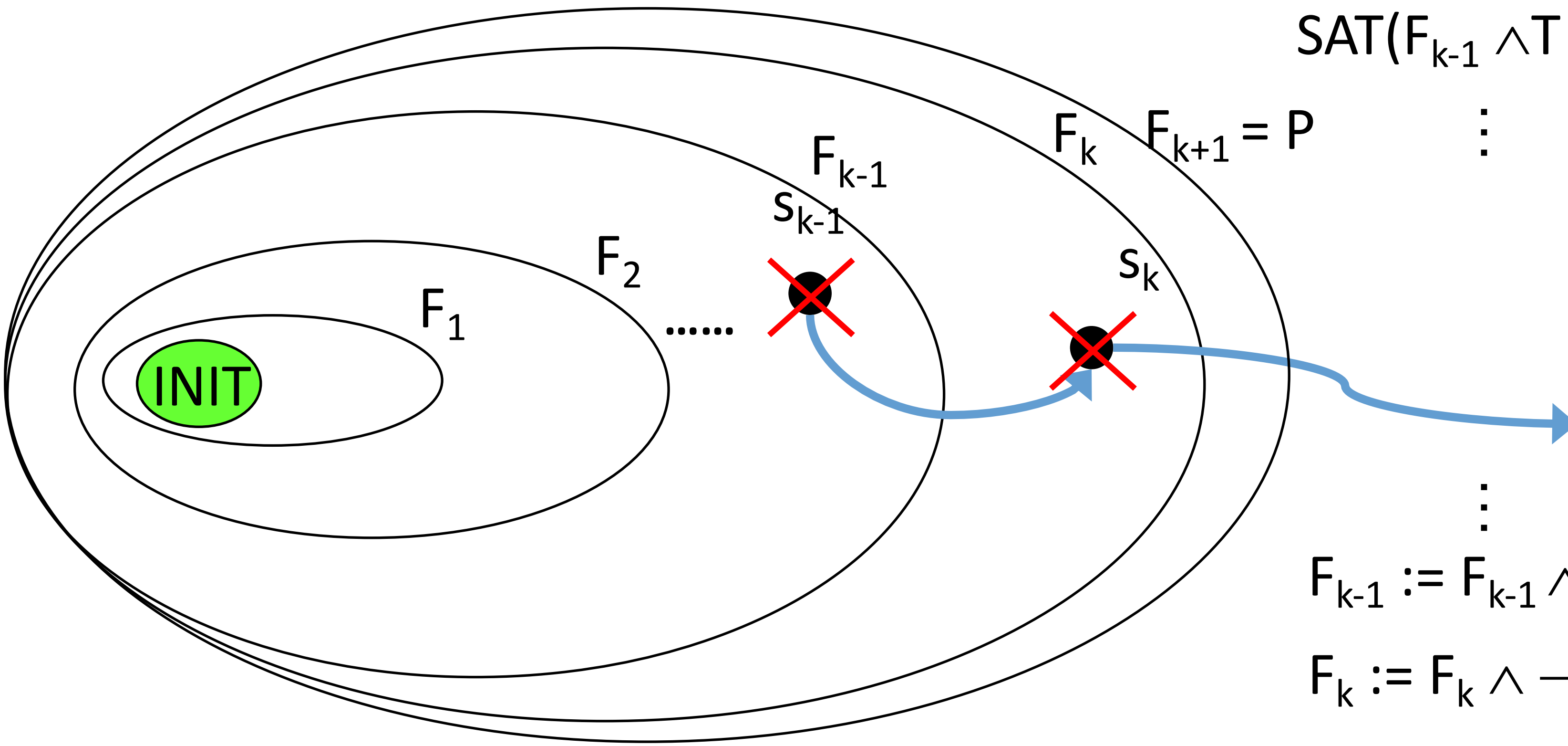
When $F_1 \wedge T \wedge s'$ becomes unsatisfiable

- s is blocked, get a "new" s* by $F_2 \wedge T \wedge \neg P'$ and try to block s*

……You get the picture ☺



General Iteration



$SAT(F_k \wedge T \wedge \neg P')$?

$SAT(F_{k-1} \wedge T \wedge s_k')$?

⋮

⋮

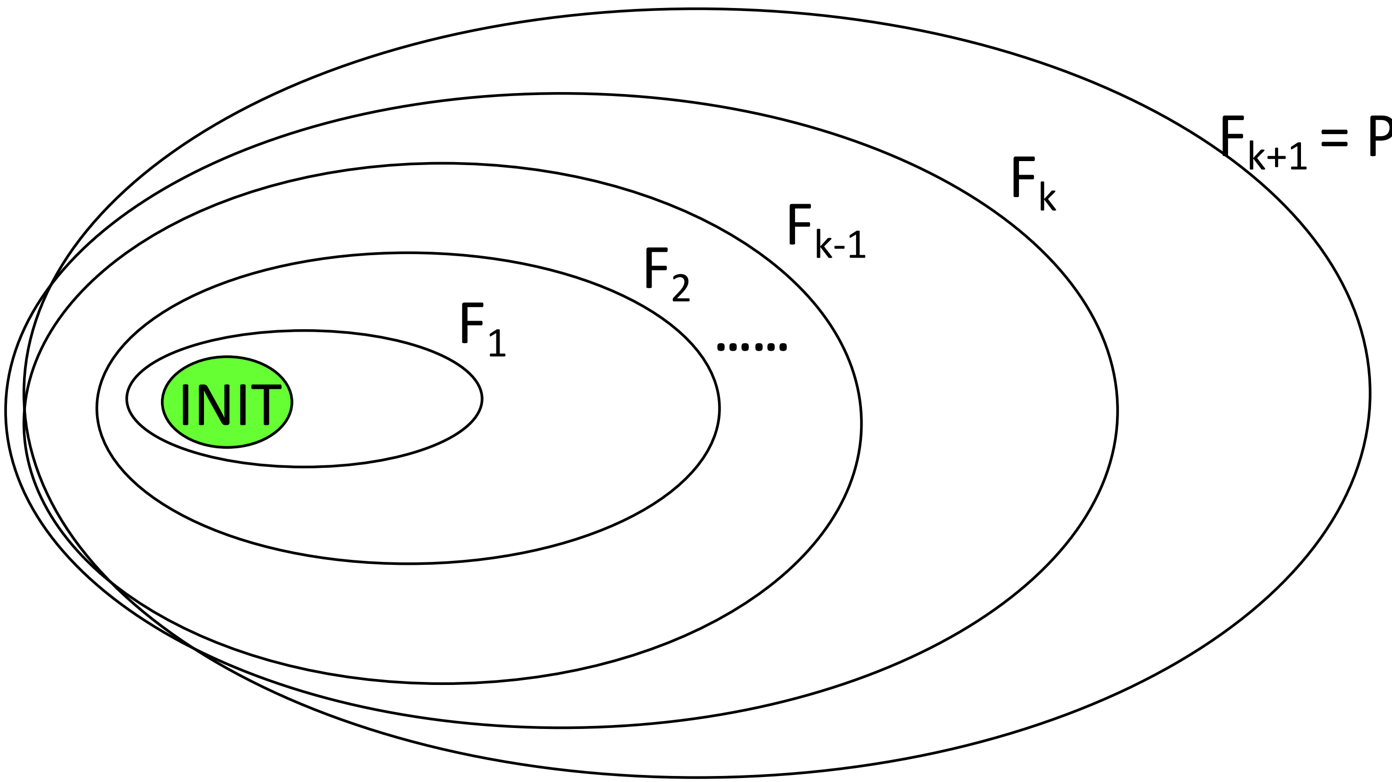
$F_{k-1} := F_{k-1} \wedge \neg s_{k-1}$

$F_k := F_k \wedge \neg s_k$

If s_k is reachable (in k steps): counterexample

If s_k is unreachable: strengthen F_k to exclude s_k

General Iteration



$\vdots$

$F_{k-1} := F_{k-1} \wedge \neg s_{k-1}$

$F_k := F_k \wedge \neg s_k$

Until $F_k \wedge T \wedge \neg P'$ is unsatisfiable, i.e. $F_k \wedge T \Rightarrow P'$

➔ We have an OARS again. Check fixpoint and increase k

IC3 - Iteration

Given an OARS $<F_0,F_1,\ldots,F_k>$, set $F_{k+1} = P$

Apply a backward search

1. Find predecessor s_k in F_k that can reach a bad state
   – $F_k \wedge T \not\Rightarrow P'$ $\quad$ ($F_k \wedge T \wedge \neg P'$ is sat)
2. If none exists, move to next iteration (check fixpoint first)
3. If exists, try to find a predecessor s_{k-1} to s_k in F_{k-1}
   – $F_{k-1} \wedge T \not\Rightarrow \neg s_k'$ $\quad$ ($F_{k-1} \wedge T \wedge s_k'$ is sat)
4. If none exists, remove s_k from F_k and go back to 3
   – $F_k := F_k \wedge \neg s_k$
5. Otherwise: Recur on (s_{k-1},F_{k-1})
   – We call $(s_{k-1},k-1)$ a “proof obligation” / “counterexample to induction”

If we reach INIT, a CEX exists

That Simple?

Looks simple

- But this “simple” does NOT work

Simple = State Enumeration

- Too many states…

Does IC3 enumerate states?

- No – removing more than one state at a time
- But, yes (when IC3 doesn’t perform well)

Generalization of a blocked state

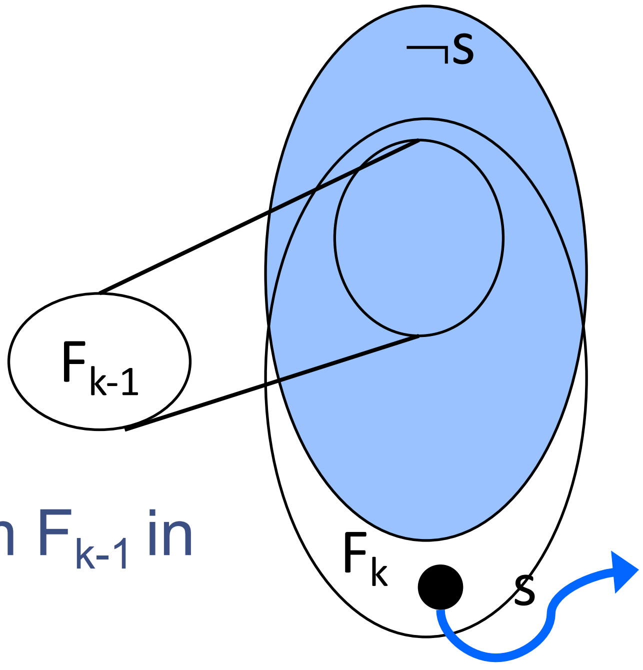
s in F_k can reach a bad state in one transition (or more)

But $F_{k-1} \wedge T \Rightarrow \neg s'$ holds

- Therefore, s is not reachable in k transitions
- $F_k := F_k \wedge \neg s$

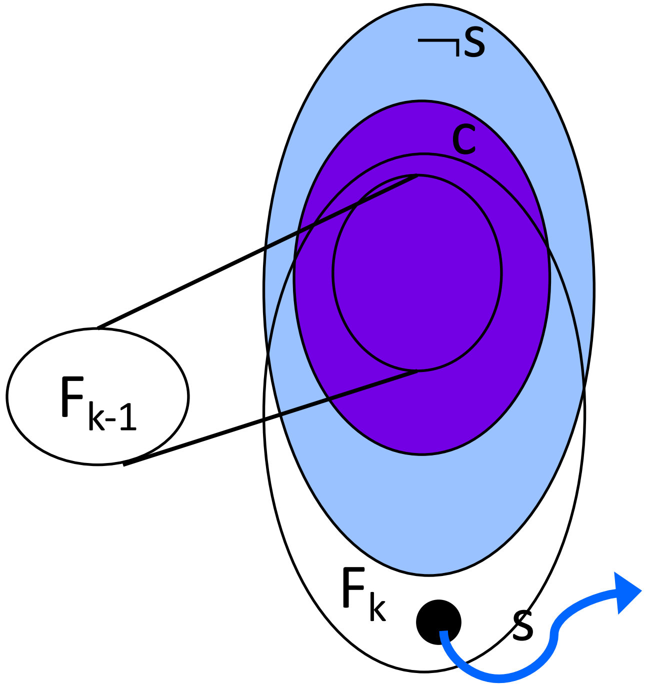
We want to generalize this fact

- s is a single state
- Goal: learn a stronger fact
  - Find a set of states, unreachable from F_{k-1} in one step



UNIVERSITY OF WATERLOO

Generalization



We know $F_{k-1} \wedge T \Rightarrow \neg s'$

And, $\neg s$ is a clause

Generalization:
Find a sub-clause $c \subseteq \neg s$ s.t.
$\mathbf{F_{k-1} \wedge T \Rightarrow c'}$ and INIT => c

- Sub clause means less literals
- Less literals implies less satisfying assignments
  - $(a \vee b)$ vs. $(a \vee b \vee c)$
- $c \Rightarrow \neg s$ i.e. c is a stronger fact

$F_k := F_k \wedge c$

- More states are removed from F_k, making it stronger/more precise (closer to R_k)

UNIVERSITY OF WATERLOO

Generalization

How do we find a sub-clause $c \subseteq \neg s$ s.t. $\mathbf{F_{k-1} \wedge T \Rightarrow c'}$**?**

Trial and Error

- Try to remove literals from $\neg s$ while $\mathbf{F_{k-1}} \wedge \mathbf{T} \wedge \neg c'$ and $\mathbf{INIT} \wedge \neg c'$ remain unsatisfiable

Use the UnSAT Core

- $\mathbf{(INIT'} \vee \mathbf{(F_{k-1} \wedge T))} \wedge s'$ is unsatisfiable
- Conflict clauses can also be used

$F_{k-1} \wedge T \wedge s'$ is UNSAT

Desired:

$c \Longrightarrow \neg s$

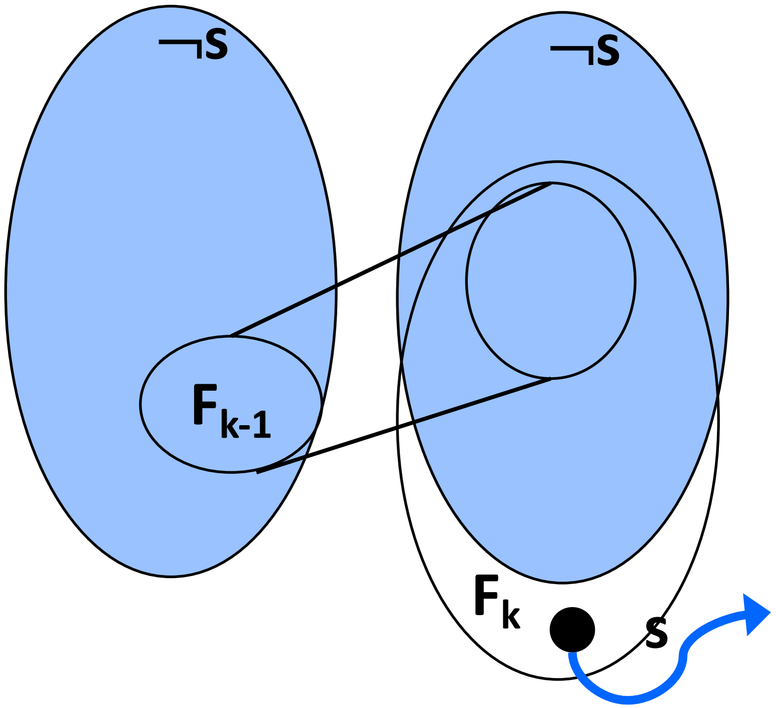
$F_{k-1} \wedge T \wedge \neg c'$ is UNSAT

Looks familiar?

Observation 1

Assume a state s in F_k can reach a bad state in a number of transitions

- Important Fact: **s is not in F_{k-1}** (!!)
  - If s was in F_{k-1} we would have found it in an earlier iteration



- Therefore: F_{k-1} => ¬s

Observation 1

Assume a state s in F_k can reach a bad state in a number of transitions

Therefore: $F_{k-1} \Rightarrow \neg s$

Assume $F_{k-1} \wedge T \Rightarrow \neg s'$ holds

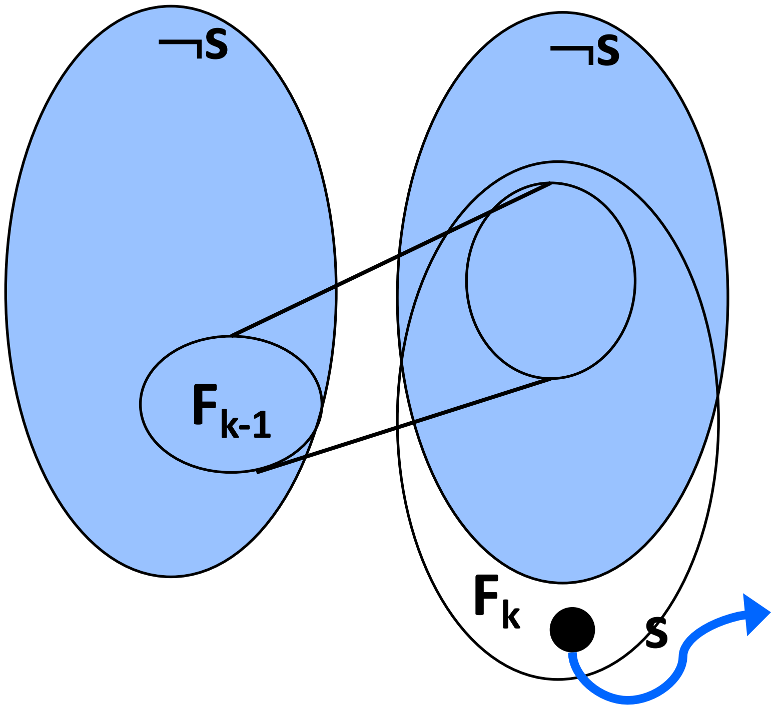
- It's blocking time…

So, this is equivalent to

$$F_{k-1} \wedge \neg s \wedge T \Rightarrow \neg s'$$

Further INIT => ¬s

- Otherwise, CEX!
  (INIT ≠> ¬s IFF s is in INIT)



- This looks familiar!
  - ¬s is inductive relative to F_{k-1}

UNIVERSITY OF WATERLOO

Inductive Generalization

We now know that ¬s is inductive relative to F_{k-1}

And, ¬s is a clause

Inductive Generalization:
Find sub-clause $c \subseteq \neg s$ s.t.

$$\mathbf{F_{k-1} \wedge c \wedge T \Rightarrow c'} \text{ (and INIT => c)}$$

- Stronger inductive fact

$F_k := F_k \wedge c$

- It may be the case that $\mathbf{F_{k-1} \wedge T \Rightarrow F_k}$ no longer holds
  - Why?

Inductive Generalization

$F_{k-1} \wedge c \wedge T \Rightarrow c'$ and $INIT \Rightarrow c$ hold

$F_k := F_k \wedge c$

c is also inductive relative to $F_{k-1}, F_{k-2}, \ldots, F_0$

- Add c to all of these sets
- For every $i \leq k$: $F_i^* = F_i \wedge c$

$\mathbf{F_i^* \wedge T \Rightarrow F_{i+1}^*}$ holds for every $i < k$

Observation 2

Assume state s in F_i can reach a bad state in a number of transitions

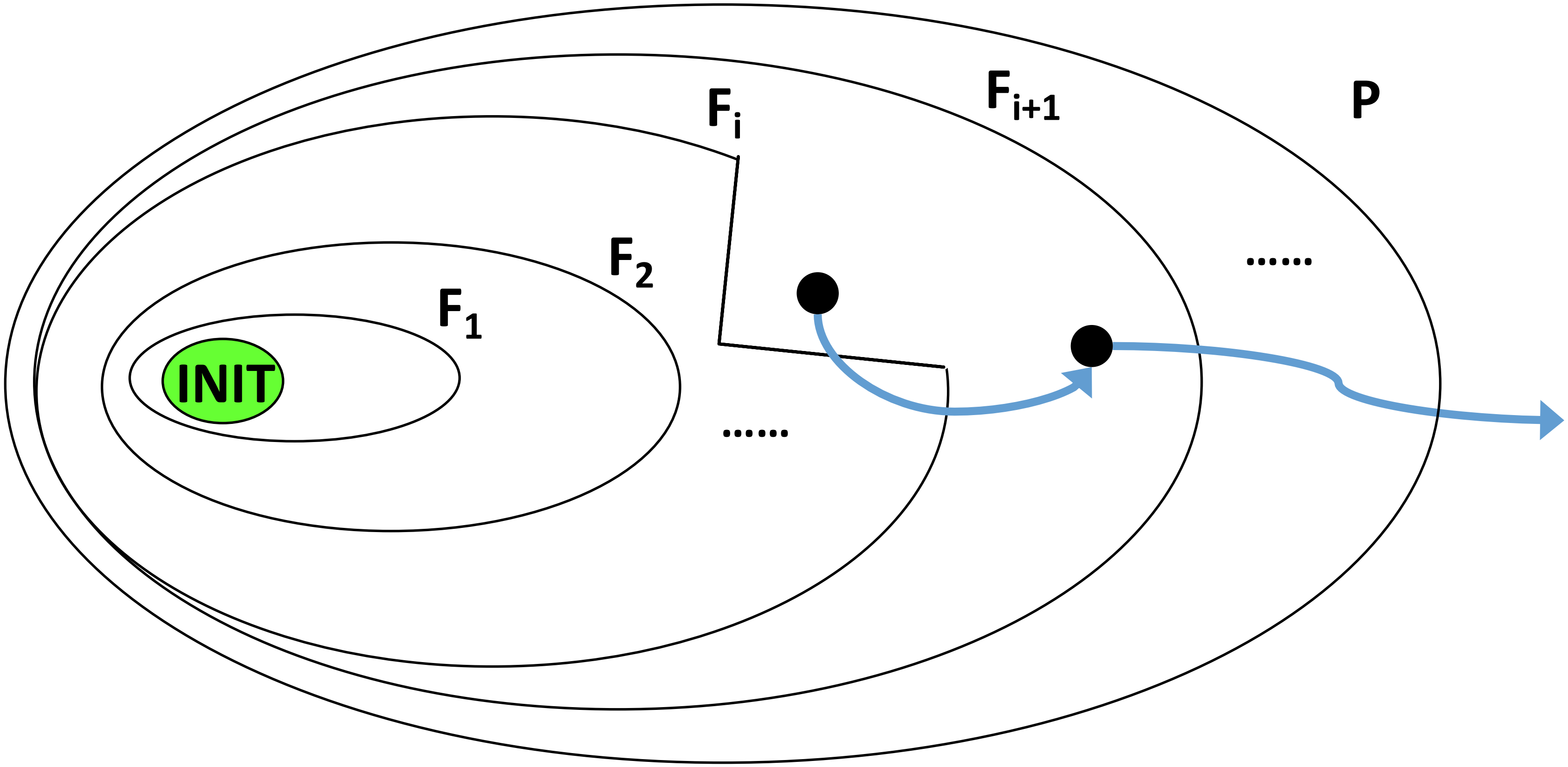
s is also in F_j for $j > i$ ($F_i \Rightarrow F_j$)

- a longer CEX may exist
- s may not be reachable in i steps, but it may be reachable in j steps

If s is blocked in F_i, it must be blocked in F_j for $j > i$

- Otherwise, a CEX exists

Push Forward



Push Forward

Suppose s is removed from F_i

- by conjoining a sub-clause c
- $F_i := F_i \wedge c$

c is a clause learnt at level i

try to push c forward for j > i

- If $F_j \wedge c \wedge T \Rightarrow c'$ holds
  - c is inductive in level j
  - $F_{j+1} := F_{j+1} \wedge c$
- Else: s was not blocked at level j > i
  - Add a proof obligation (s,j)
  - If s is reachable from INIT in j steps, CEX!

UNIVERSITY OF WATERLOO

Generalizing Predecessor

Suppose s_{k-1} is a predecessor obtained by $F_{k-1} \wedge T \wedge s_k'$

- New proof obligation

Try to generalize s_{k-1} to a set of states (cube m) such that

$$m \Longrightarrow \exists V' \,.\, F_{k-1} \wedge T \wedge s_k'$$

- Drop a literal from s_{k-1} and use ternary simulation to check whether $F_{k-1} \wedge T \wedge s_k'$ evaluates to true under current assignment

UNIVERSITY OF WATERLOO

Recursive Blocking Stage in IC3

```
// Find a counterexample, or strengthen the inductive trace
// s.t. F_N ⇒ ¬s holds
IC3_recBlockCube(s, N)
    Add(Q, (s, N))
    while ¬Empty(Q) do
        (s, k) ← Pop(Q)
        if (k = 0) return "Counterexample"
        if (F_k ⇒ ¬s) continue
        if (F_k-1 ∧ Tr ∧ s') is SAT
            t ← generalized predecessor of s
            Add(Q, (t, k-1))
            Add(Q, (s, k))
        else
            ¬t ← generalize ¬s by inductive generalization (to
                                                             level m≥k)
            add ¬t to F_m
            if (m<N) Add(Q, (s, m+1))
```

Pushing stage in IC3

```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c’)
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return “Proof” // F_k is a safe inductive invariant
```

IC3 – Key Ingredients

Backward Search

- Find a state s that can reach a bad state in a number of steps
- [lifting: generalize s to a set of states]
- s may not be reachable (over-approximations)

Block a State

- Do it efficiently, block more than s
  - Generalization / Inductive generalization

Push Forward

- An inductive fact at frame i, may also be inductive at higher frames
- If not, a longer CEX may be found

Pushing to the Top with K-induction

Arie Gurfinkel
Electrical and Computer Engineering
University of Waterloo

joint work with Alexander Ivrii (IBM)

UNIVERSITY OF WATERLOO

Agenda

IC3 is one of the most powerful algorithms for model checking safety properties

Very active area of research:

- A. Bradley: *SAT-Based Model Checking Without Unrolling.* VMCAI 2011
  (IC3 stands for "Incremental Construction of Inductive Clauses for Indubitable Correctness")

- N. Eén, A. Mishchenko, R. Brayton: *Efficient implementation of property directed reachability.* FMCAD 2011
  (PDR stands for "Property Directed Reachability")

  …

- In this talk, I present a new IC3-based algorithm, called QUIP
  (QUIP stands for "a QUest for an Inductive Proof")

A brief preview of Quip

Quip extends IC3 by allowing for

- *A wider range of conjectures (proof obligations)*
  - Designed to push already existing lemmas more aggressively
  - Allows to push a given lemma by learning additional *supporting* lemmas
    (and hopefully to compute an inductive invariant faster)

- *Forward reachable states*
  - Explain why a lemma cannot be pushed
  - Allows to keep the number of proof obligations under control

These are integrated into a single algorithmic procedure

The experimental results look good

A quick review of IC3/PDR

Input:
- A safety verification problem (Init, Tr, Bad)

Output:
- A counterexample (if the problem is UNSAFE),
- A safe inductive invariant (if the problem is SAFE)
- Resource Limit

Main Data-structures:
- A current working level N
- An *inductive trace*
- A set of *proof obligations*

Inductive Trace

Let $F_0, F_1, F_2, \ldots, F_\infty$ be conjunctions of lemmas (in practice, clauses).
We say that $F_0, F_1, F_2, \ldots, F_\infty$ is an *inductive trace* if:

(1) $F_0 = INIT$
(2) $F_0 \Rightarrow F_1 \Rightarrow F_2 \Rightarrow \ldots \Rightarrow F_\infty$ *(monotone)*
(3) $F_1 \supseteq F_2 \supseteq \ldots \supseteq F_\infty$ as sets of lemmas *(s. monotone)*
(4) $F_i \wedge TR \Rightarrow F_{i+1}'$ for $i \geq 0$ (including $F_\infty \wedge Tr \Rightarrow F_\infty'$). *(inductive)*

Remarks:
This definition is slightly different from the original definition:

- the sequence $F_0, F_1, F_2, \ldots$ is conceptually *infinite* (with $F_i = T$ for all sufficiently large i)
- we add F_∞ as the last element of the trace (as suggested in PDR)

Each F_i over-approximates states that are reachable in i steps or less (in particular, F_∞ contains all reachable states)

Proof Obligations in IC3

A *proof obligation* in IC3 is a pair (s, i), where

- s is a (generalized) cube over state variables
- i is a natural number (called *level*)

We say that (s, i) is *blocked* (or that *s is blocked at level i*) if $F_i \Rightarrow \neg s$.
Given a proof obligation (s, i), IC3 attempts to *strengthen* the inductive trace in order to block it.

Remarks:
In IC3, s is identified with a *counterexample-to-induction* (*CTI*)

If (s, i) is a proof obligation and $i \geq 1$, then (s, i-1) is already blocked

All proof obligations are managed via a *priority queue*:

- Proof obligations with smallest level are considered first
- (additional criteria for tie-breaking)

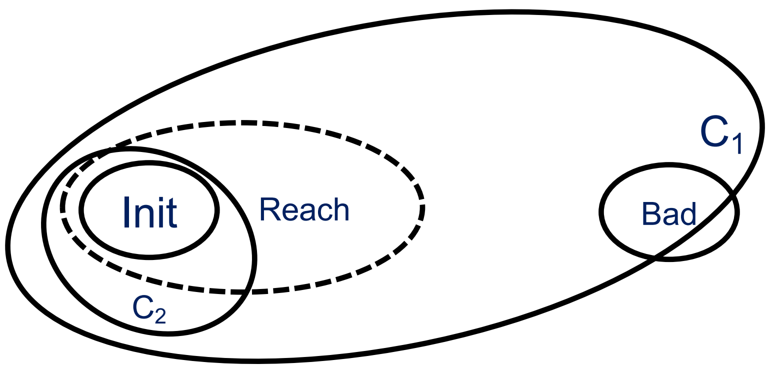
UNIVERSITY OF WATERLOO

Towards improving IC3 (1)

IC3 is an excellent algorithm! So, what do we want?

We want *more control* on which lemmas to learn:

- Each lemma in the inductive trace is neither an over-approximation nor an under-approximations of reachable states (a lemma in F_k only over-approximates states reachable within k steps):
    - IC3 may learn lemmas that are *too weak* (ex. C_1) – prune less states
    - IC3 may learn lemmas that are *too strong* (ex. C_2) – cannot be in the inductive invariant



Towards improving IC3 (2)

We want to know if *an already existing lemma* is *good* (in F_∞) or *bad* (e.g., C_2 from before):

- Avoid periodically pushing bad lemmas
- Ideally, we also want to prune less useful lemmas

We want to *prioritize reusing already discovered lemmas* over learning of new ones:

- When the same cube s is blocked at different levels, usually different lemmas are discovered
  - Although, IC3 partially addresses this using pushing (and other optimizations)
- Use the same lemma to block s (at the expense of deriving additional supporting lemmas)
  - Although, in general different lemmas are of different “quality” and having some choice may be beneficial

Immediate improvement: unlimited pushing

```
// Push each clause to the highest possible frame ~~up to N~~
IC3_Push_Unlimited()
    for k = 1 .. do
        for c ∈ F_k \ F_k+1 do
            if (F_k ∧ Tr ⇒ c')
                add c to F_k+1
        if (F_k = F_k+1)
            F_∞ ← F_k
        if (F_∞ ⇒ ¬Bad)
            return "Proof" // F_∞ is a safe inductive invariant
```

Claim: after pushing F_∞ represents a *maximal inductive subset* of all lemmas discovered so far

Remark: the idea to compute maximal inductive invariants is suggested in PDR but claimed to be ineffective. In our implementation, "unlimited pushing" leads to ~10% overall speed up.

Pushing is Useful

Why pushing is useful:

- During the execution of IC3, the sets F_i are incrementally strengthened, and so it may happen that $F_k \wedge TR \Rightarrow c'$, even though this was not true at the time that c was discovered

Why pushing is good:

- By pushing c from F_k to F_{k+1}, we make F_k *more inductive* (and if F_k becomes equal to F_{k+1}, then F_k becomes an inductive invariant)
- Suppose that $c \in F_k$ blocks a proof obligation (s, k). By pushing c from F_k to F_{k+1}, we also block the proof obligation $(s, k+1)$

- Pushing Clauses = Improving Convergence = Reusing old lemmas for blocking bad states

What Happens when Pushing Fails

Why pushing may fail: suppose that $c \in F_k \setminus F_{k+1}$ but $F_k \wedge TR$ does not imply c'. *Why?*

There are two alternatives:

1. c is a valid over-approximation of states reachable within $k+1$ steps, but F_k is not strong enough to imply this
   - We can strengthen the inductive trace so that $F_k \wedge TR \Rightarrow c'$ becomes true

2. c is **NOT** a valid over-approximation of states reachable within $k+1$ steps
   - There is a real *forward reachable* state r that is excluded by c
   - c has no chance to be in the safe inductive invariant
   - c is a *bad* lemma

A similar reasoning is used in:
Z. Hassan, A. Bradley, F. Somenzi: *Better Generalization in IC3*. FMCAD 2013

Two interdependent ideas

1. Prioritize pushing existing lemmas
   - Given a lemma $c \in F_k \setminus F_{k+1}$, we can add $(\neg c, k+1)$ as a *may-proof-obligation*
     - May-proof-obligations are "nice to block", but do not need to be blocked
   - If $(\neg c, k+1)$ can be blocked, then c is pushed to F_{k+1}
   - If $(\neg c, k+1)$ cannot be blocked, then we discover a *concrete reachable state* r that is excluded by c and that *explains* why c cannot be inductive

2. Discover and use new forward reachable states
   - These are an *under-approximation* of forward reachable states
   - Given a reachable state, all the existing lemmas that exclude it are *bad*
     - Bad lemmas are never pushed
   - Reachable states may show that certain may-proof-obligations cannot be blocked
   - Reachable states may be used when generalizing lemmas
   - Conceptually, computing new reachable states can be thought of as *new* Init states

Quip

Input:
- A safety verification problem (Init, Tr, Bad)

Output:
- A counterexample (if the problem is UNSAFE),
- A safe inductive invariant (if the problem is SAFE)
- Resource Limit

Main Data-structures:
- A current working level N
- An *inductive trace* (same as IC3)
- A set of *proof obligations* (*similar* to IC3)
- A set R of *forward reachable states*

Proof Obligations in Quip

A proof obligation in Quip is a triple (s, i, p), where

- s is a (generalized) cube over state variables
- i is a natural number
- $p \in \{may, must\}$

Remarks:

- As in IC3, if (s, i, p) is a proof obligation and $i \geq 1$, then $(s, i-1)$ is assumed to be already blocked
- As in IC3, all proof obligations are managed via a priority queue:
  - Proof obligations with *smallest level* are considered first
  - In case of a tie, proof obligations with *smallest number of literals* are considered first
  - (additional criteria for tie-breaking)
- Have a "*parent map*" from a proof obligation to its parent proof obligation
  - $parent(t) = s$ if $(t, k-1, q)$ is a predecessor of (s, k, p)
  - In fact, this is usually done in IC3 as well (for trace reconstruction)

Recursive Blocking Stage in Quip (1)

1. Each time that we examine a proof obligation (s, k, p), check whether s intersects a reachable state $r \in R$

2. Discover new reachable states when possible
   - Claim: if s intersects $r \in R$ and if parent(s) exists, then there exists a reachable state r' that intersects parent(s)
     - Indeed, **ALL** states in s lead to a state in parent(s)
     - Therefore r leads to a state in parent(s) as well
   - A similar idea is present in: C. Wu, C. Wu, C. Lai, C. Huang: *A counterexample-guided interpolant generation algorithm for SAT-based model checking*. TCAD 2014

3. When (s, k, p) is blocked by an inductive lemma $\neg t$, add $(t, k+1, \mathit{may})$ as a new proof obligation
   - Push $\neg t$ to F_{k+1} instead of blocking $(s, k+1)$

4. Clear all proof obligations if their number becomes too large (important, not in pseudocode)

Recursive Blocking Stage in Quip (2)

```
// Find a reachable state r∈s, or strengthen the inductive trace
s.t. F_N ⇒ ¬s
Quip_recBlockCube(s, N, q)
    Add(Q, (s, N, q))
    while ¬Empty(Q) do
        (s, k, p) ← Pop(Q)
        if (k = 0) && (p = must) return "Counterexample"
        if (k = 0) && (p = may)
            find a state r one-step-reachable from Init,
                such that r intersects parent(s)
            add r to R; continue
        if (F_k ⇒ ¬s) continue
        if (s intersects some state r∈R) && (p = must) return
                                                  "Counterexample"
        if (s intersects some state r∈R) && (p = may)
            if parent(s) exists, find a state r' one-step-reachable
                                                           from r,
              such that r' intersects parent(s)
            add r' to R; continue
//  -- continued on the next slide --
```

Recursive Blocking Stage in Quip (3)

```
Quip_recBlockCube(s, N, p)
//  -- continued from the previous slide --
        if (F_{k-1} ∧ Tr ∧ s') is SAT
                t ← generalized predecessor of s
                Add(Q, (t, k-1, p))
                Add(Q, (s, k, p))
        else
            ¬t ← generalize ¬s by inductive
                    generalization (to level m≥k)
            add ¬t to F_m
            if (m<N)
                if (t = s)  Add(Q, (t, m+1, p))
                else       Add(Q, (t, m+1, may))
                      // attempt to block t (not s)
```

UNIVERSITY OF WATERLOO

Experiments: IC3 vs. Quip on HWMCC'13 and '14

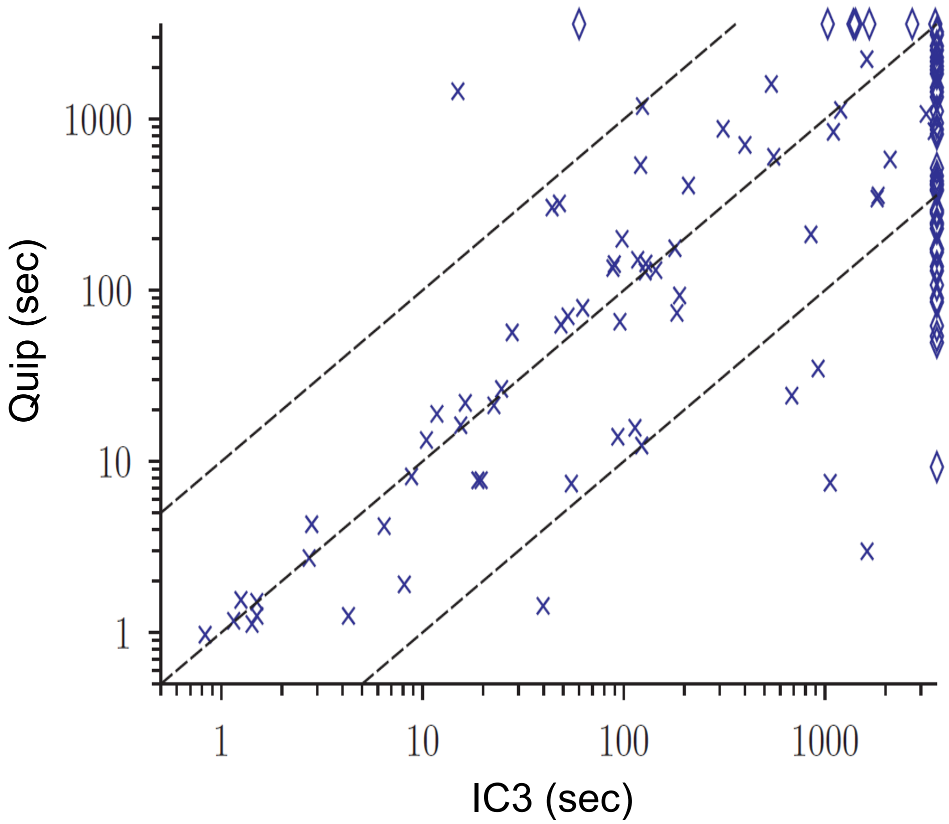
	UNSAFE solved	UNSAFE time	SAFE solved	SAFE time
`IC3`	22 (2)	52,302	76 (7)	137,244
`Quip`	32 (12)	20,302	99 (30)	69,590

Experimental results on the instances solved by either `IC3` or `Quip` separated into unsafe and safe instances. The numbers in parentheses represent the unique solves. The times are in seconds.

- Implemented in IBM formal verification tool *Rulebase-Sixthsense*
- Data for 140 instances that were not trivially solved by preprocessing but could be solved either by IC3 or Quip within 1-hour
- Detailed results at http://arieg.bitbucket.org/quip

UNIVERSITY OF WATERLOO

Experiments: IC3 vs. Quip on HWMCC'13 and '14



- Data for 140 instances from prev slide

UNIVERSITY OF WATERLOO