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")

- **and** show how QUIP can be extended with k-induction

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

Problem: 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.

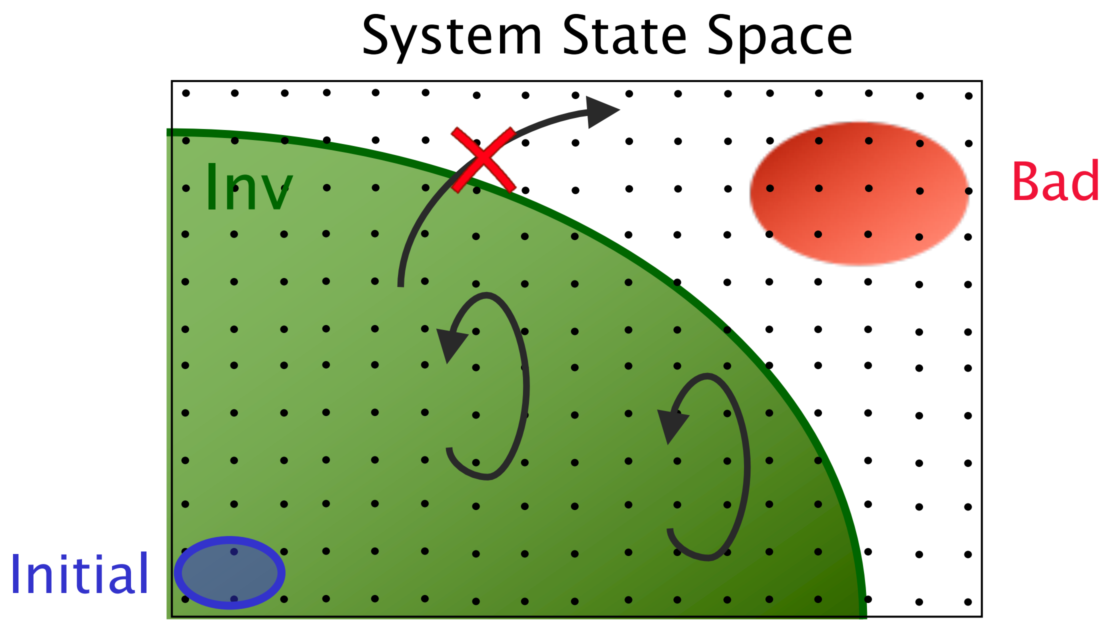
$$Init(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \wedge Bad(X_N) \not\Rightarrow \perp$$

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

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

$$Inv \Rightarrow \neg Bad \qquad \boxed{\text{Safe}}$$

Inductive Invariants



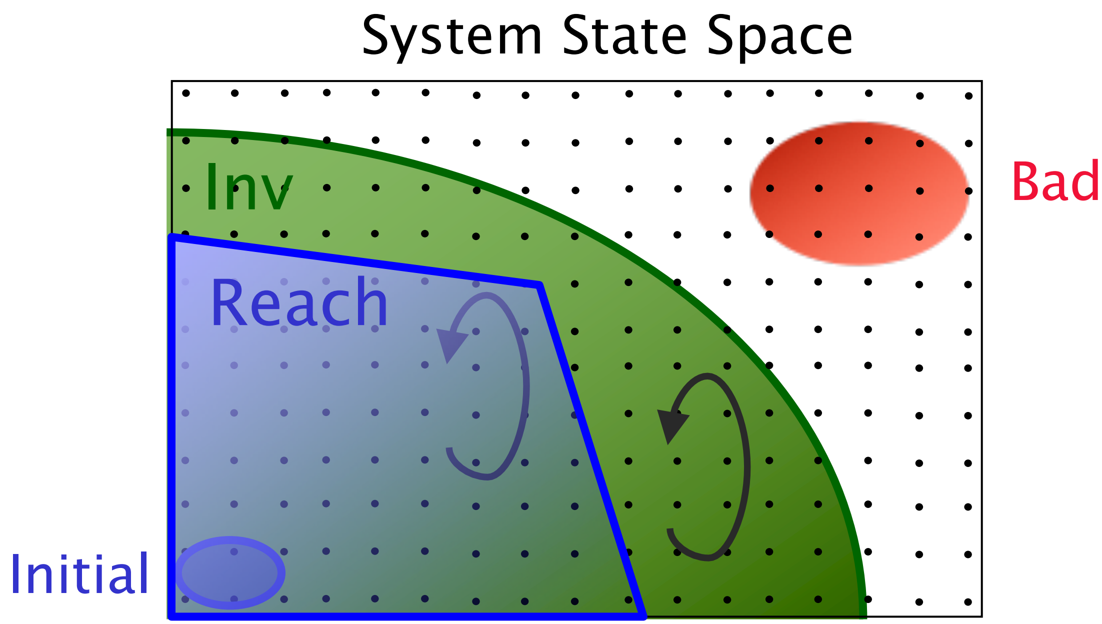
System S is safe iff there exists an **inductive invariant** Inv:

- 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

UNIVERSITY OF WATERLOO

Inductive Invariants



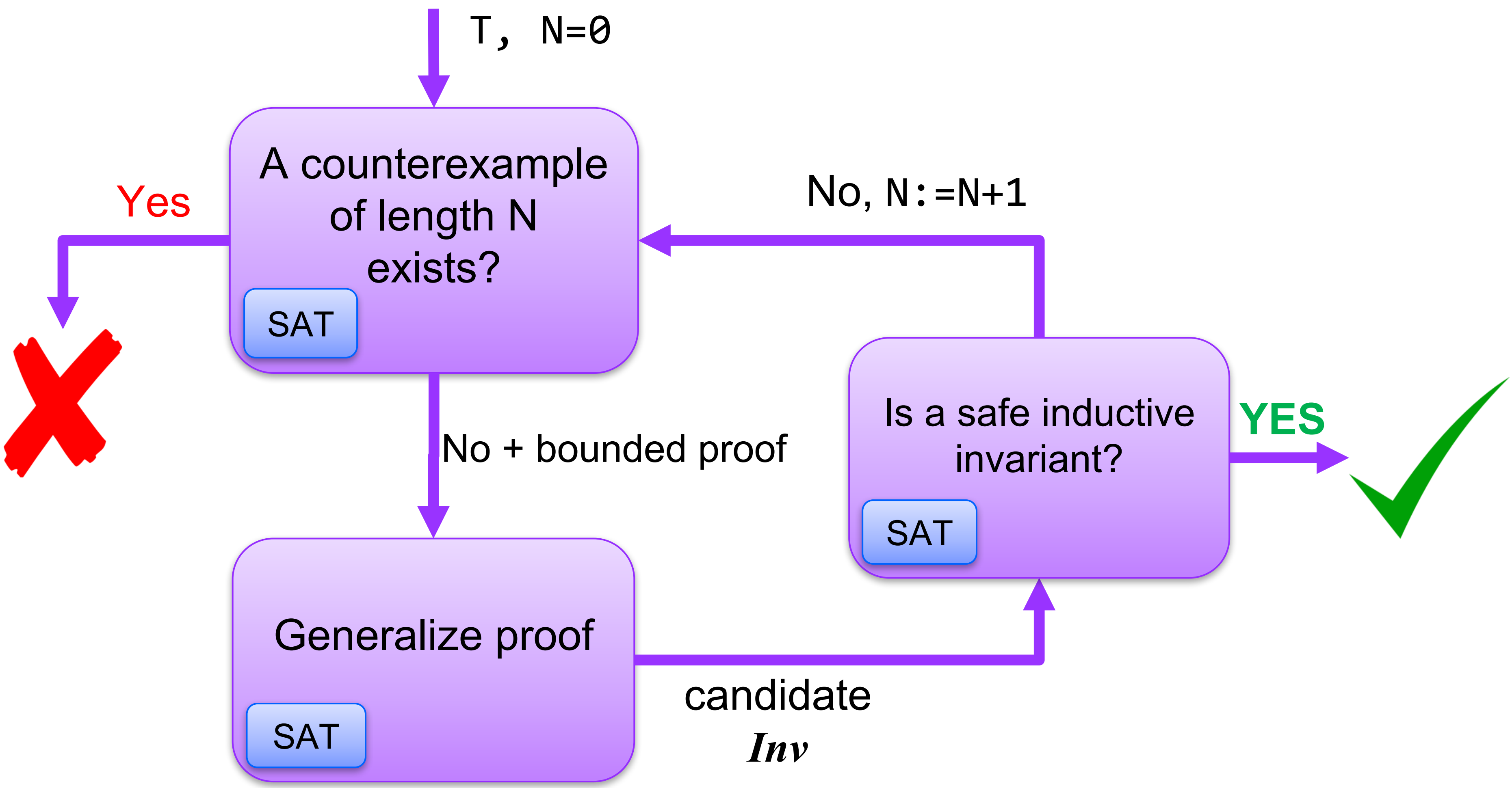
System S is safe iff there exists an **inductive invariant** Inv:

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

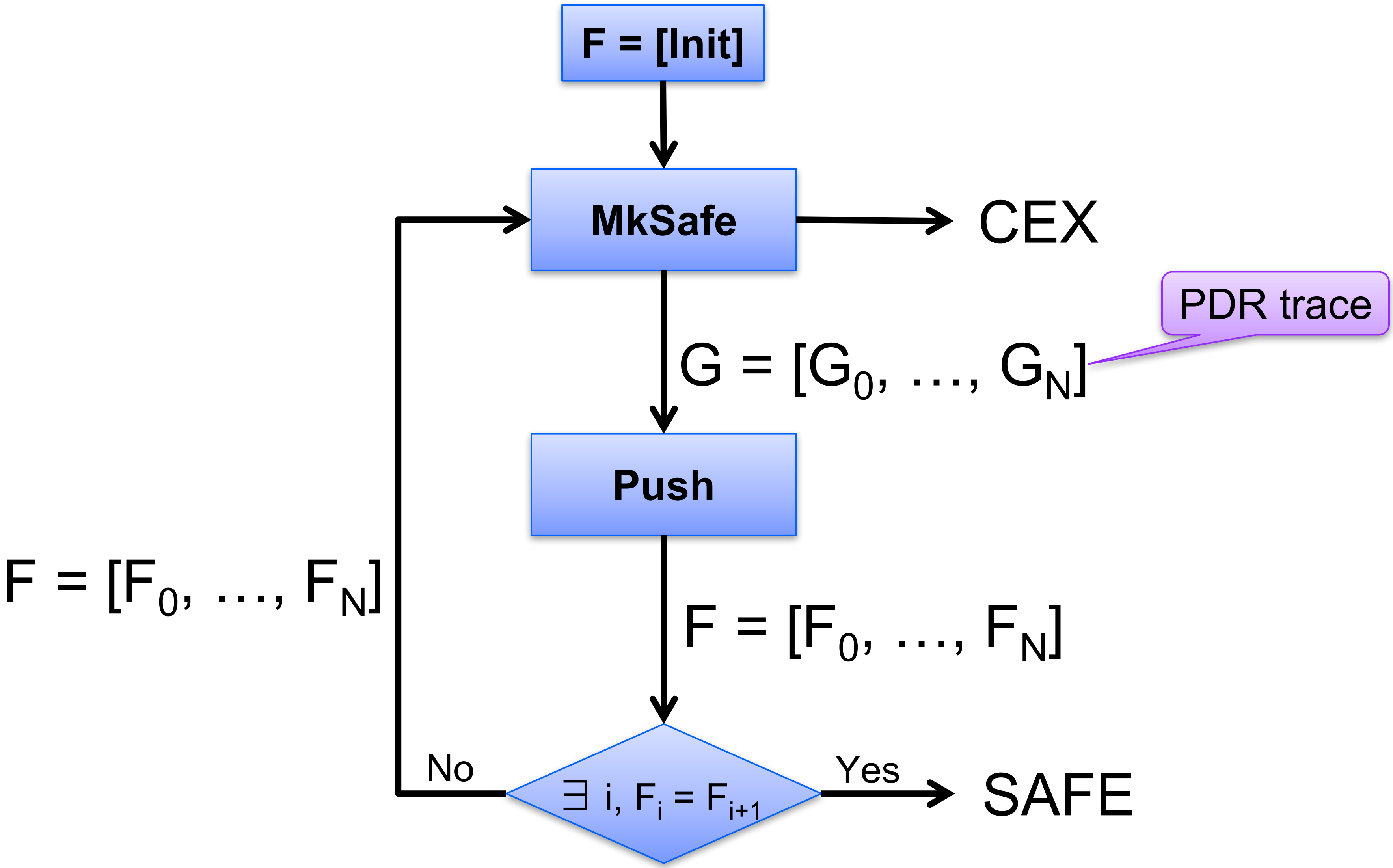
i.e., if s ∈ Inv and s⇝t then t ∈ Inv

System S is **safe** if Reach ∩ Bad = ∅

Generalizing from Bounded Proofs



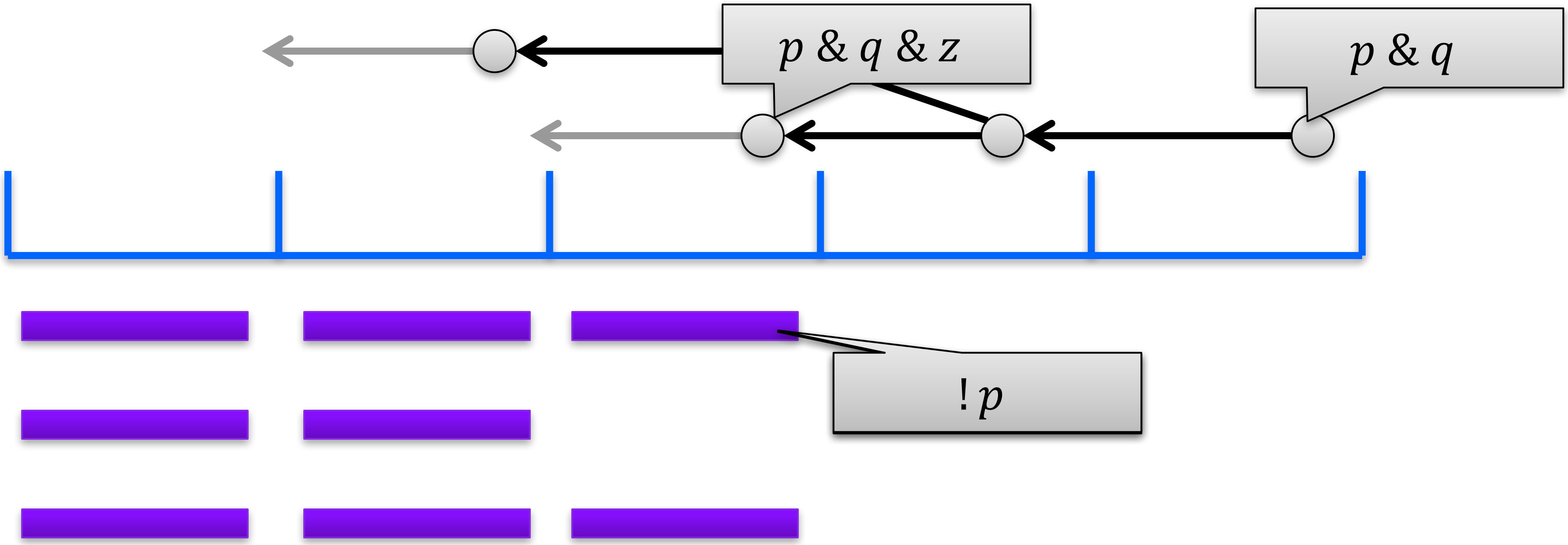
IC3/PDR

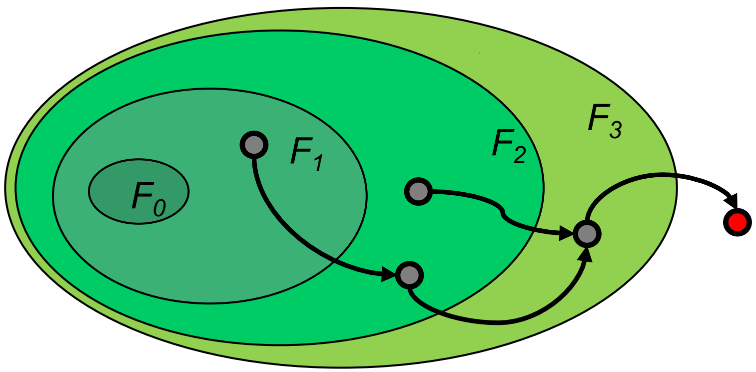


UNIVERSITY OF WATERLOO

IC3/PDR In Pictures: MkSafe

MkSafe

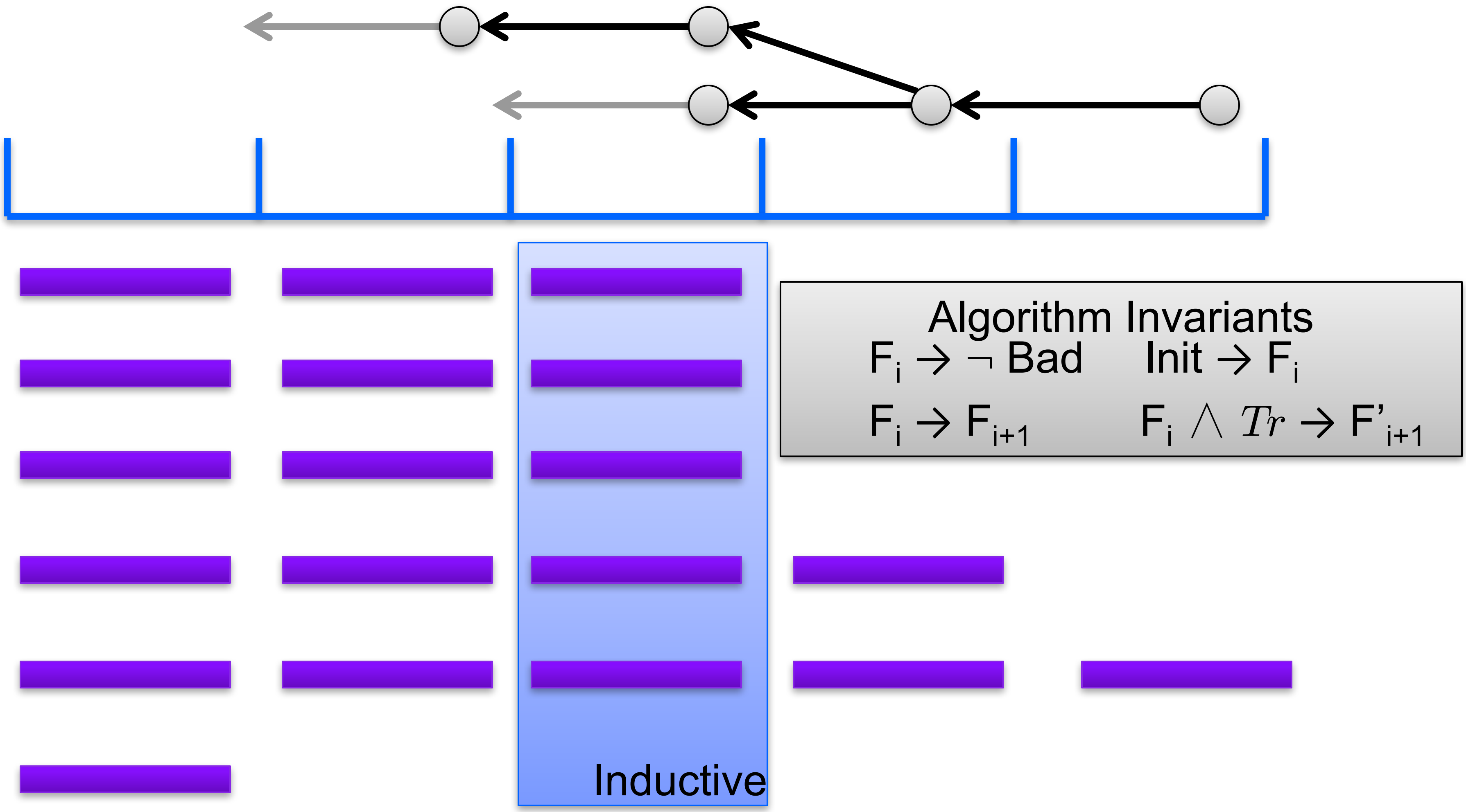




UNIVERSITY OF WATERLOO

IC3/PDR in Pictures: Push

Push



Algorithm Invariants

$F_i \rightarrow \neg$ Bad Init $\rightarrow F_i$

$F_i \rightarrow F_{i+1}$ $F_i \wedge Tr \rightarrow F'_{i+1}$

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)*

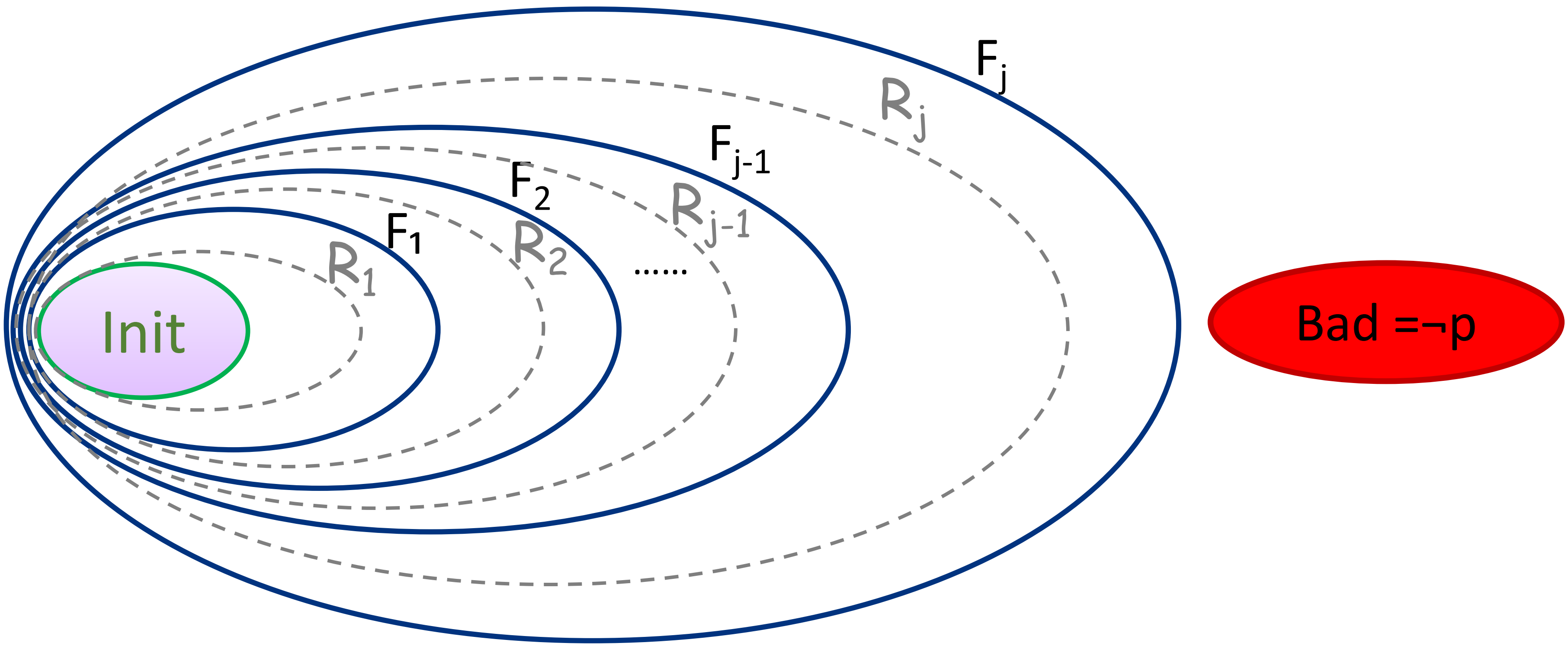
(4) $F_i \wedge TR \Rightarrow F_{i+1}'$ for $i \geq 0$ (including $F_\infty \wedge Tr \Rightarrow F_\infty'$). *(inductive)*

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)

Safe Monotone Inductive Trace



- F_i over-approximates the states that are reachable in at most i steps
- If $F_{j+1} \Rightarrow F_j$ then F_j is an inductive invariant

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)

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))
```

UNIVERSITY OF WATERLOO

Pushing stage in IC3

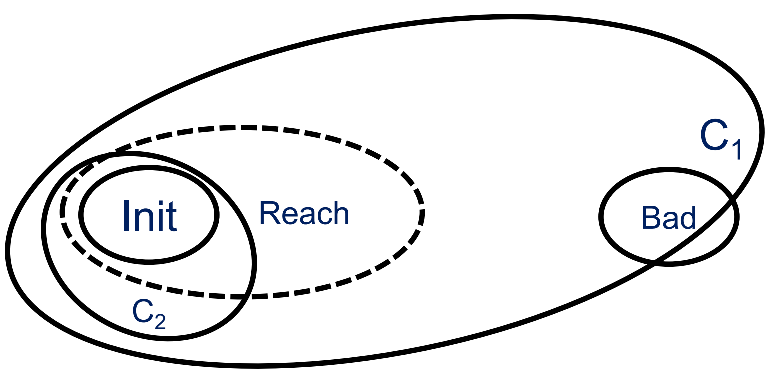
```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c')
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return "Proof" // F_k is a safe inductive invariant
```

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

UNIVERSITY OF WATERLOO

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:

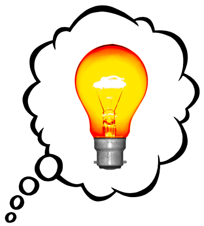
1. c is a valid over-approximation of states reachable within $k+1$ steps, but F_k is not strong enough to imply this
   - We can strengthen the inductive trace so that $F_k \wedge TR \Rightarrow c'$ becomes true

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

UNIVERSITY OF WATERLOO

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, *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 --
```

UNIVERSITY OF WATERLOO

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)
```

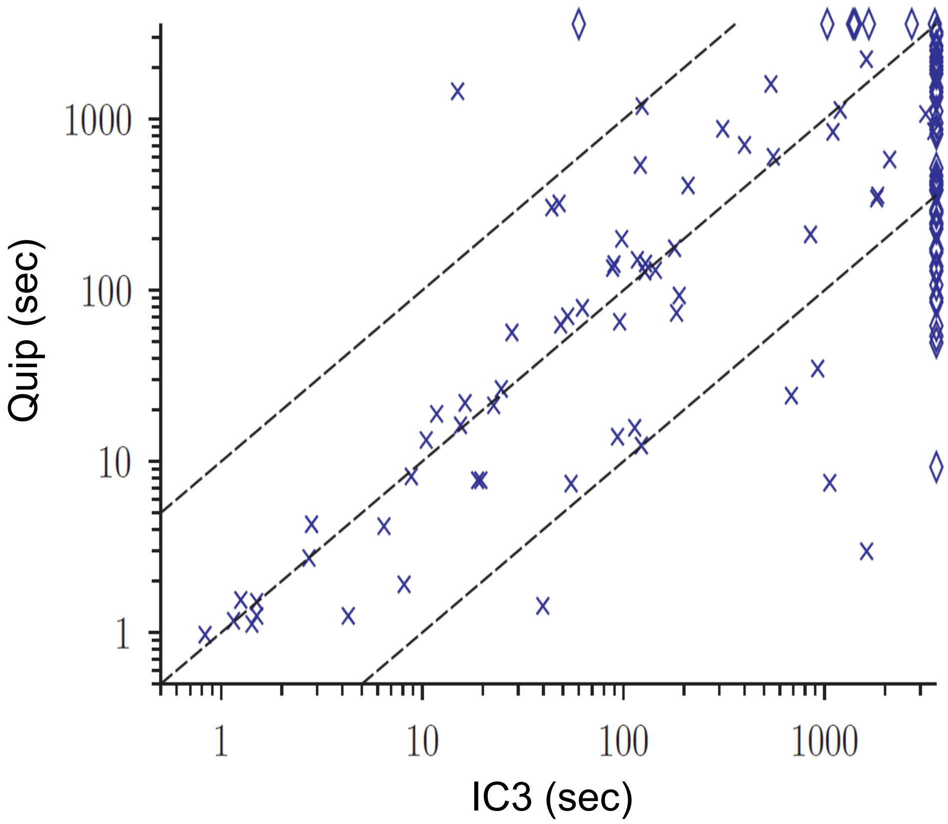
Experiments: IC3 vs. Quip on HWMCC'13 and '14

	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

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



- Data for 140 instances from prev slide

Quip – future work

- Improve handling of forward reachable states (both for performance and memory)

- Generalize forward reachable states

- Incorporate these ideas with other known IC3 developments
  - Abstraction-Refinement:
    Y. Vizel, O. Grumberg, S. Shoham: *Lazy abstraction and SAT-based reachability in hardware model checking*. FMCAD 2012
  - Lemma generalization:
    Z. Hassan, A. Bradley, F. Somenzi: *Better Generalization in IC3*. FMCAD 2013

- Experiment with other ways to combine the ideas into a full algorithm

- Lift Quip to more general domains

K-Induction without Unrolling

FMCAD'17

Arie Gurfinkel
Alexander Ivrii

K-induction Principle

[SSS2000]

Induction:

$$\frac{P(s_0) \qquad \forall i: P(s_i) \Rightarrow P(s_{i+1})}{\forall i: P(s_i)}$$

k-step Induction:

$$\frac{P(s_{0..k-1}) \qquad \forall i: P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

SAT-based Model Checking with K-induction

Let the k-unrolling of transition relation be defined as:

$U_k = T^{<0>} \wedge T^{<1>} \wedge \ldots \wedge T^{<k-1>}$

Use SAT solver to check validity of two formulas:

- Base case:

$$I^{<0>} \wedge U_{k-1} \Rightarrow P^{<0>} \ldots P^{<k-1>}$$

- Induction step:

$$U_k \wedge P^{<0>} \ldots P^{<k-1>} \Rightarrow P^{<k>}$$

If both are valid, then P is true in all the reachable states

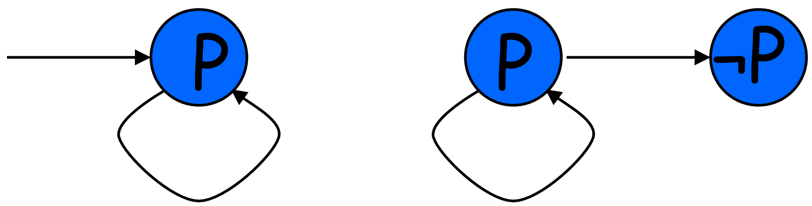
If the base case is invalid, there is a counterexample

If the induction step is invalid, increase k and try again

Simple path assumption

Unfortunately, k-induction is not complete

- some properties are not k-inductive for any k
- for example,



Simple path restriction:

- There is a path to ¬P iff there is a *simple* path to ¬P (path with no repeated states)

Complete k-induction with simple paths

Let simple($s_{0..k}$) be defined as:

- $\forall$ i,j in 0..k : (i ≠ j) $\Rightarrow s_i \neq s_j$

k-induction over simple paths:

$$\frac{P(s_{0..k-1}) \qquad \forall i: \text{simple}(s_{0..k}) \wedge P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

Must hold for k large enough, since a length of the longest simple path is bounded (recurrence diameter)

Terminology: k-invariants and k-induction

- φ is an invariant if it holds on all reachable states

- φ is a k-invariant if it holds on all states reachable in up to k steps:
  $Init(X_0) \wedge Tr(X_0, X_1) \wedge \ldots \wedge Tr(X_{N-1}, X_N) \Rightarrow \varphi(X_N)$ for all $0 \le N \le k$

- φ is a k-inductive invariant if φ is a (k-1)-invariant, and
  $\varphi(X_0) \wedge Tr(X_0, X_1) \wedge \ldots \wedge \varphi(X_{k-1}) \wedge Tr(X_{k-1}, X_k) \Rightarrow \varphi(X_k)$

- k-induction states that if φ is a k-inductive invariant, then φ is an invariant

UNIVERSITY OF WATERLOO

k-induction vs IC3

- k-induction and IC3 have complementary strengths, both theoretically and practically

- Can we understand both algorithms in a common way?
  - we present an IC3-like algorithm to show whether a given safety property is k-inductive

- Can we devise an effective algorithm combining the two?
  - we show how IC3 can be extended with k-inductive reasoning – with minor modifications of the IC3-algorithm

k-induction vs IC3

k-induction and IC3 algorithms are heavily used for unbounded model checking in both hardware and software domains

Theoretically, the two algorithms have complementary strengths:

- k-induction (with loop-free constraints) is stronger than 1-induction
- IC3 derives new lemmas to strengthen the property

Practically (on hardware benchmarks)

- k-induction is mostly successful for small values of k (up to 10)
- IC3 solves many more properties than k-induction
- However, there are properties that can be proved by k-induction with low value of k, but IC3 “gets lost”

k-induction without unrolling

- We present K-IND – an algorithm to decide whether a (k-1)-invariant formula φ is k-inductive

- K-IND returns:
  - φ is k-inductive, OR
  - φ is not k-inductive (and a counterexample to k-induction)

- Highlights:
  - Does not unroll the transition relation

  - Guarantees loop-free constraints without introducing expensive unique-state constraints

K-IND: for k=3 and without loop-free constraints

- Let φ be 2-invariant
  - No Init-state can reach a $\neg\varphi$-state in 0, 1 or 2 steps
  - Taking 0, 1 or 2 successive predecessors of a $\neg\varphi$-state cannot get to an Init-state

- We want to check/determine whether φ is 3-inductive invariant

  - $\varphi(X_0) \wedge Tr(X_0, X_1) \wedge \varphi(X_1) \wedge Tr(X_1, X_2) \wedge \varphi(X_2) \wedge Tr(X_2, X_3) \Rightarrow \varphi(X_3)$?

- Equivalently, we want to check whether the following formula is satisfiable

  - $\varphi(X_0) \wedge Tr(X_0, X_1) \wedge \varphi(X_1) \wedge Tr(X_1, X_2) \wedge \varphi(X_2) \wedge Tr(X_2, X_3) \wedge \neg\varphi(X_3)$

- Equivalently, φ is a 3-inductive invariant if and only if we cannot start from a $\neg\varphi$-state and find 3 successive predecessors satisfying φ

K-IND: demonstration

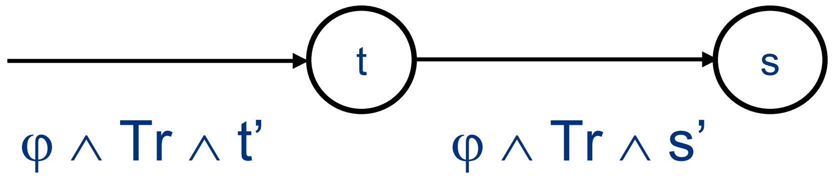
- We use an IC3-like algorithm to check satisfiability:
  $\varphi(X_0) \wedge Tr(X_0, X_1) \wedge \varphi(X_1) \wedge Tr(X_1, X_2) \wedge \varphi(X_2) \wedge Tr(X_2, X_3) \wedge \neg\varphi(X_3)$

- For simplicity, assume that $\varphi = \neg s$, where s is a cube over registers

- t is a predecessor of s
- u is a predecessor of t
- v is a predecessor of u
- We have found 3 successive predecessors: φ is not 3-inductive
  - Moreover, if v is an Init-state, then we have a counter-example to φ

K-IND: demonstration

- We use an IC3-like algorithm to check satisfiability:
  $\varphi(X_0) \wedge Tr(X_0, X_1) \wedge \varphi(X_1) \wedge Tr(X_1, X_2) \wedge \varphi(X_2) \wedge Tr(X_2, X_3) \wedge \neg\varphi(X_3)$
- For simplicity, assume that $\varphi = \neg s$, where s is a cube over registers



- Suppose now that t has no predecessors satisfying φ
- As in IC3, we can learn a lemma $\neg t$ that explains why t is not reachable
  - All SAT queries are made relative to the same frame
  - All lemmas "hold for all frames"
  - No need re-enqueue discharged proof obligations

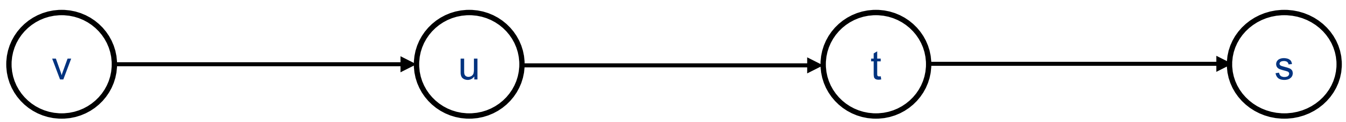
K-IND: demonstration

- Putting it all together:
  - Keep a stack of proof obligations $\{(s, i)\}$, initially $\{(\neg\varphi, 0)\}$
    - Where i represents the "depth" rather than "level"
  - Keep a set of lemmas G, initially empty
  - Iteratively:
    - Take the top proof obligation t and make a predecessor query
      $G \wedge \varphi \wedge Tr \wedge t'$
    - If found a depth-3 predecessor u and u is $Init$, return "Counter-Example"
    - Else if found a depth-3 predecessor, return "Counter-Example to 2-induction"
    - Else if found a predecessor u, add a new proof obligation $(u, i\text{-}1)$
    - Else adds $\neg t$ to G
    - If $G \Rightarrow \neg s$, return "Blocked"

- As in IC3, we can generalize $\neg t$ as long as $Init \Rightarrow \neg t$ and $G \wedge \varphi \wedge Tr \wedge \neg t \Rightarrow \neg t'$
  - in this case, the algorithm may return "Blocked" even when φ is not 3-inductive
  - But, G is still an inductive invariant proving φ

K-IND: loop-free constraints

- To integrate simple-path constraints, add (the negations of) all parent states to the predecessor queries



$\neg u \wedge \neg t \wedge \neg s \wedge \varphi \wedge Tr \wedge u'$

$\neg s \wedge \varphi \wedge Tr \wedge s'$

$\neg t \wedge \neg s \wedge \varphi \wedge Tr \wedge t'$

- Here s, t, u, v are generalized proof obligations (= sets of states)

K-IND: relatively k-inductive

- Let F be 0-invariant ($Init \Rightarrow F$)
- φ is a k-inductive invariant relative to F if φ is a k-invariant, and

$$\varphi(X_0) \wedge F(X_0) \wedge Tr(X_0, X_1) \wedge \ldots \wedge \varphi(X_{k-1}) \wedge F(X_{k-1}) \wedge Tr(X_{k-1}, X_k) \Rightarrow \varphi(X_k)$$

- K-IND can be extended to determine if φ is k-inductive relative to F

K-IND: experimental results

In practice (on hardware benchmarks, and without loop-free constraints):

- K-IND solves a few more properties than k-induction
  - Due to lemma generalization

- When solved by both, k-induction is usually faster than K-IND
  - Exactly in the same way as BMC is usually faster than IC3 when looking for counterexamples

KIC3: K-Inductive IC3

- Related Work: PD-KIND
  - “Property-Directed k-Induction”, Dejan Jovanović and Bruno Dutertre, FMCAD’2016
  - Variant of IC3/PDR based on k-induction
  - Effective on SMT benchmarks
  - Requires unrolling transition relation for validating k-inductive queries
    - A direct implementation of PD-KIND for hardware does not scale
  - Has strongly inspired our solution

- We present KIC3
  - A framework extending IC3 with k-inductive reasoning
  - Integrates k-induction into IC3 with minor modifications of the IC3-framework
  - But does not fully incorporate loop-free constraints

k-inductive blocking

- Given a proof obligation (s, i), we can attempt to block s using k-induction relative to F_{i-1}
  - Can choose any $k \leq i$

- Let's suppose that $F_{i-1} \wedge \neg s \wedge Tr \wedge s'$ is satisfiable, and t is a predecessor of s in F_{i-1}

- IC3:
  - Adds a new proof obligation $(t, i-1)$
  - Proceeds to checking whether $F_{i-2} \wedge \neg t \wedge Tr \wedge t'$ is satisfiable

- k-inductive blocking (for $k > 1$):
  - Adds a new proof obligation (t, i)
  - Proceeds to checking whether $F_{i-1} \wedge \neg t \wedge Tr \wedge t'$ is satisfiable

Comparison of KIC3-blocking and IC3-blocking

- What happens if both $F_{i-2} \wedge \neg t \wedge Tr \wedge t'$ and $F_{i-1} \wedge \neg t \wedge Tr \wedge t'$ are UNSAT?
  - IC3 learns a lemma valid (at least) to F_{i-1}
  - k-inductive blocking learns a lemma valid (at least) to F_i
  - Lemmas learned by k-inductive blocking are in general of "higher quality"

- What happens if $F_{i-2} \wedge \neg t \wedge Tr \wedge t'$ is UNSAT, while $F_{i-1} \wedge \neg t \wedge Tr \wedge t'$ is SAT?
  - IC3 stops blocking (t, i-1) and returns to blocking (s, i)
  - k-inductive blocking continues blocking predecessors of t
  - If k-inductive blocking succeeds blocking the topmost proof-obligation (s, i)
    - It may learn more lemmas, but all of these lemmas are valid (at least) to F_i

- Recall that using k-induction to block (s, i) may return "counterexample-to-k-induction"
  - Simple solution: fall back to blocking (s, i) using IC3
  - Inspired by PD-KIND: block this counterexample-to-k-induction using IC3* and continue blocking (s, i) using k-induction
  -

*or we can again use m-inductive blocking for a suitable value of m

Alternative ways to think about KIC3-blocking

- k-inductive blocking is IC3 with a slightly different strategy for managing proof obligations
  - IC3 schedules proof obligations at the lowest level they are unknown
  - k-inductive blocking may schedule proof obligations to higher levels instead

- k-inductive blocking can be thought of as abstraction
  - Given a proof obligation (t, j), IC3 checks whether $\neg t$ is inductive relative to F_{j-1}
  - However, any abstraction of F_{j-1} can be used instead
  - For example, using only lemmas from F_{i-1} closely corresponds to KIC3

Which states to block using KIC3-blocking?

- For the experiments, we modified the procedure for recursive blocking of $(\neg Bad, i)$:
  - First, use k-inductive blocking of $(\neg Bad, i)$
  - If unsuccessful, block $(\neg Bad, i)$ as usual

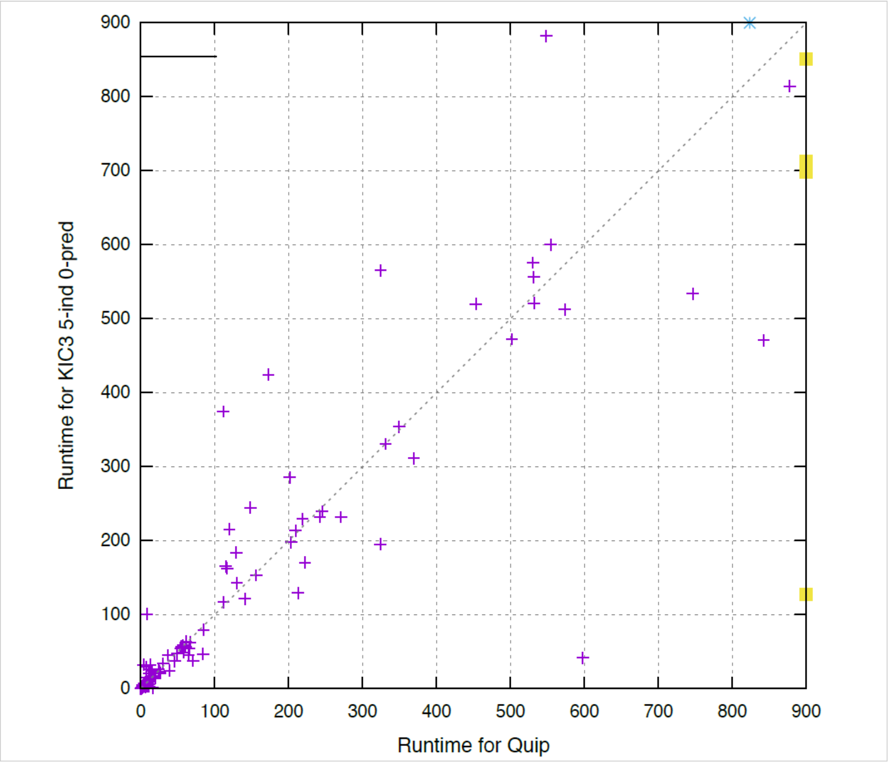
- Can also use k-inductive blocking during the pushing stage of the IC3 algorithm
  - Directly inspired by PD-KIND

Experimental Results

- Implemented in IBM's formal verification tool on top of Quip
- 238 single-property designs from HWMCC'15
  (all the designs that are not solved by simple logic synthesis, but are solved either by at least one configuration considered)
- Experimented with k = the induction depth, and m = the number of counterexamples to k-induction blocked using IC3
- 15-minutes time-limit (per property)

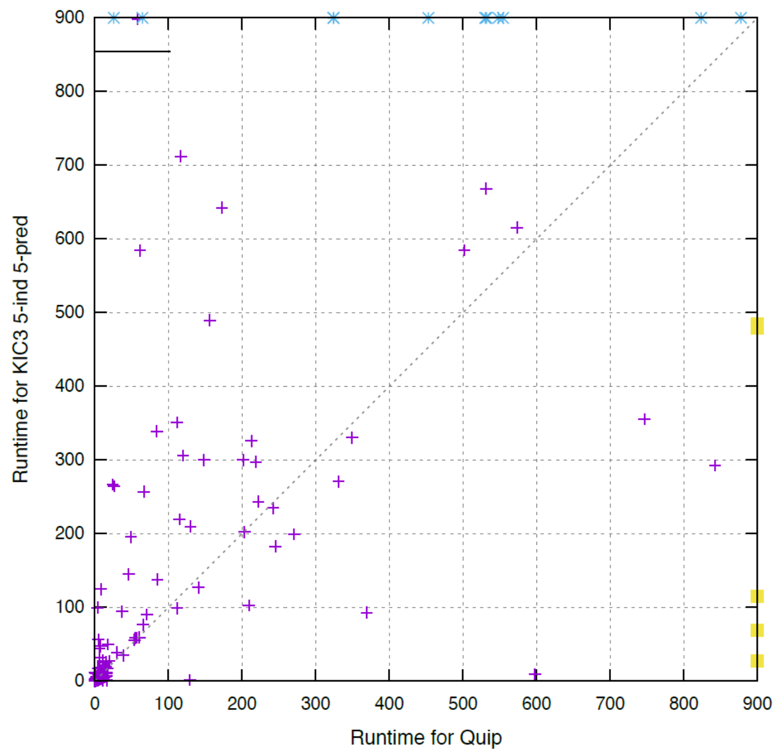
Experimental Results

- Increasing k, while fixing m=0:
  - Slightly in favor of using k-induction
  - Runtimes are highly correlated
- The scatter plot has k=5 and m=0
  - Points above diagonal = wins for IC3
  - Points below diagonal = wins for KIC3
  - IC3 solves 230 properties in 52,776 s
  - KIC3 solves 233 properties in 51,695 s
- Similar observations on proprietary designs and larger time-limits



Experimental Results

- Increasing m, while fixing k:
  - Significantly degrades performance
  - Runtimes are less correlated
- The scatter plot has k=5 and m=5
  - Points above diagonal = wins for IC3
  - Points below diagonal = wins for KIC3
  - IC3 solves 230 properties in 52,776 s
  - KIC3 solves 224 properties in 57,864 s
- Similar observations on proprietary designs and larger time-limits



Conclusion

IC3 is a great algorithm for hardware Model Checking

- but, it can still be improved

QUIP: QUest for an Inductive Proof

- aggressively push existing lemmas
- enlarge initial state by computing reachable states
- use reachable states to prune bad lemmas

KIC3: IC3 and k-induction

- k-induction without unrolling (and without simple path constraints)
- integrates easily into IC3 framework
- expensive, hard to control when to apply

IC3 is a great framework to explore MC strategies