Pushing to the Top

FMCAD’15

Arie Gurfinkel
Alexander Ivrii

Safety Verification

Consider a verification problem (Init, Tr, Bad)

The problem is UNSAFE if and only if there exists a *path from an Init-state to a Bad-state*, that is

$\mathrm{Init}(X_0) \wedge \mathrm{Tr}(X_0, X_1) \wedge \ldots \wedge \mathrm{Tr}(X_{N-1}, X_N) \wedge \mathrm{Bad}(X_N)$ is satisfiable for some N

The problem is SAFE if and only if there exists a *safe inductive invariant* G, that is

$\mathrm{Init}(X) \Rightarrow G(X)$

$G(X) \wedge \mathrm{Tr}(X, X') \Rightarrow G(X')$

$G(X) \Rightarrow \neg \mathrm{Bad}(X)$

Agenda

IC3 is one of the most powerful algorithms for proving safety

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 work we 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 considering

- *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

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* (explained in a moment)
- A set of *proof obligations* (explained in a moment)

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) $F_0 \Rightarrow F_1 \Rightarrow F_2 \Rightarrow \ldots \Rightarrow F_\infty$

(3) $F_1 \supseteq F_2 \supseteq \ldots \supseteq F_\infty$ as sets of lemmas

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

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 i sufficiently large)
    - 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 the IC3 algorithm, s is identified with a *counterexample-to-induction* (and called a *CTI*)
- If (s, i) is a proof obligation and $i \geq 1$, then $(s, i-1)$ is assumed to be already blocked
- All proof obligations are managed via a *priority queue*:
  - Proof obligations with smallest level are considered first
  - (additional criteria for tie-breaking)

IC3 algorithm

The next two slides briefly describe the two main stages of IC3

- The *recursive blocking stage*
- The *pushing stage*

We omit many important details, and concentrate on *how* IC3 works rather than *why* (there are many excellent references for this)

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

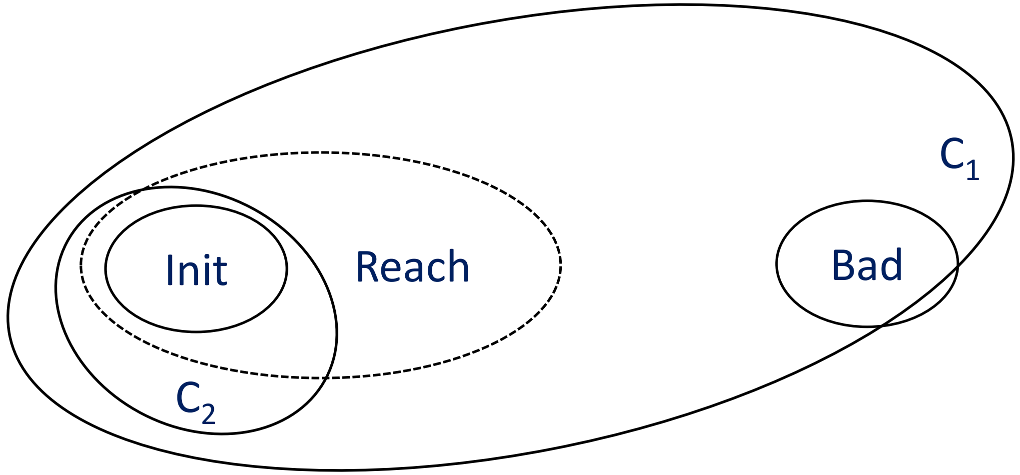
```
// Push each clause to the highest possible frame up to N
IC3_Push()
    for k = 1 .. N-1 do
        for c ∈ F_k \ F_{k+1} do
            if (F_k ∧ Tr ⇒ c')
                add c to F_{k+1}
        if (F_k = F_{k+1})
            return "Proof" // F_k is a safe inductive invariant
```

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* (ex. 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
  - Though, IC3 partially addresses this using pushing (and other optimizations)
- Use the same lemma to block s (at the expense of deriving additional supporting lemmas)
  - Though, 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.

More about pushing (1)

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

More about pushing (2)

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 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
   - Try to 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)
```

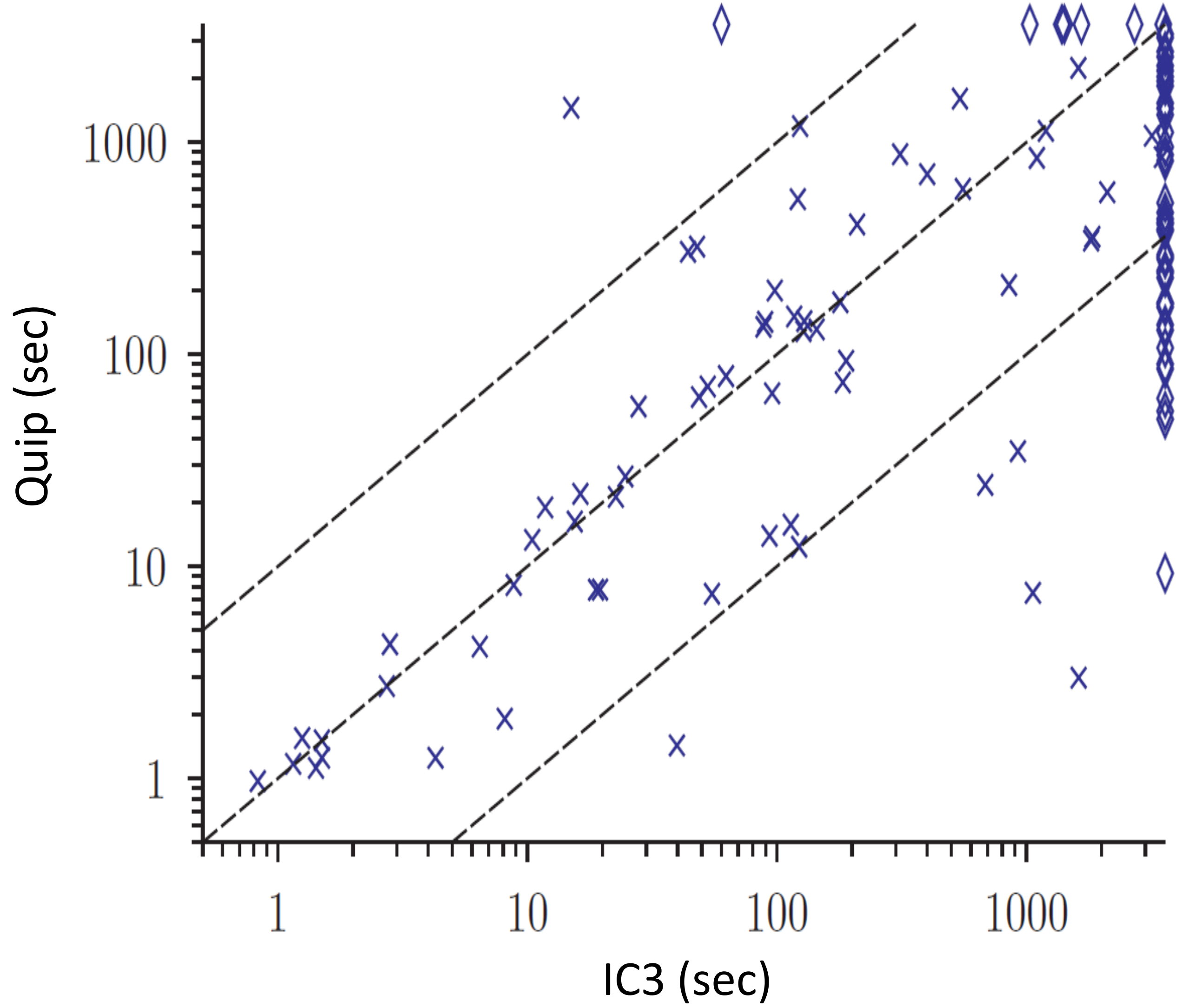
Experiments: IC3 vs. Quip on HWMCC'13 and '14

	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 last slide

Quip – alternative implementations

There are many ways to combine basic algorithmic steps to a complete algorithm. We have tried the following variants (more details in the paper).

Reset-Free Variant:

- Keep (negation of) every lemma as a proof obligation (at the corresponding level)
- Can avoid the external pushing stage altogether!

Garbage-Collection Variant:

- Periodically remove all bad lemmas from the system

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

Thank You!!!

P.S.: We hope the title of the paper now makes sense.

P.P.S.: Can you guess what are google images for Push to the Top?

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

TABLE II. DATA ON REACHABLE STATES DISCOVERED BY QUIP

# reach. states	0–10	11 – 100	101 – 1K	1K – 10K	10K – 50K
# instances	42	19	29	32	9
# unique solved	1	1	10	22	8