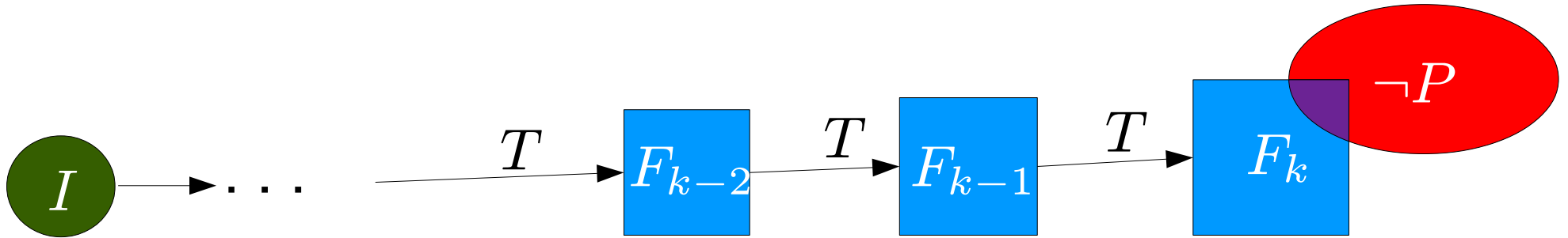

DIFTS 2014

Comparing different variants of IC3 for Hardware Model Checking

Alberto Griggio, Marco Roveri
Fondazione Bruno Kessler – Trento, Italy

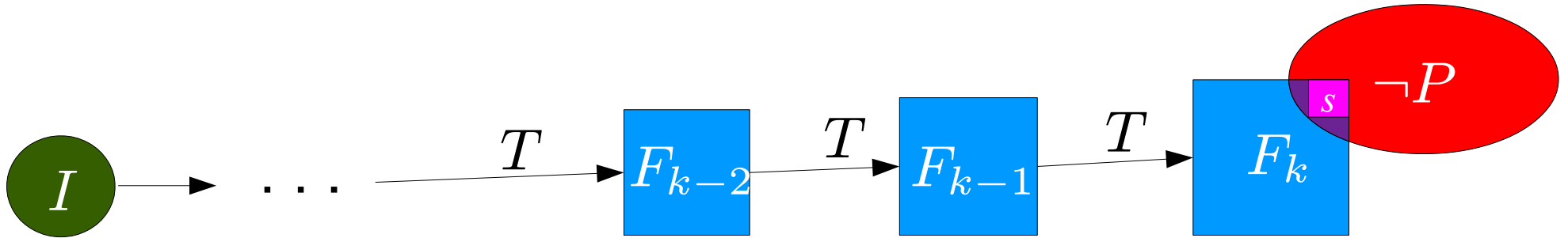
- **IC3** very successful **SAT-based** model checking algorithm
 - Very effective on **hardware designs**
- **Many variants and optimizations** proposed in the literature
 - Different implementations, different sets of benchmarks
 - Difficult to compare/evaluate
- **This work: a comprehensive evaluation of IC3 variants**
 - A **single implementation** platform (nuXmv)
 - A **common set of benchmarks** (HWMCC '11, '12, '13)
 - Some interesting results

A (very) high level view of IC3



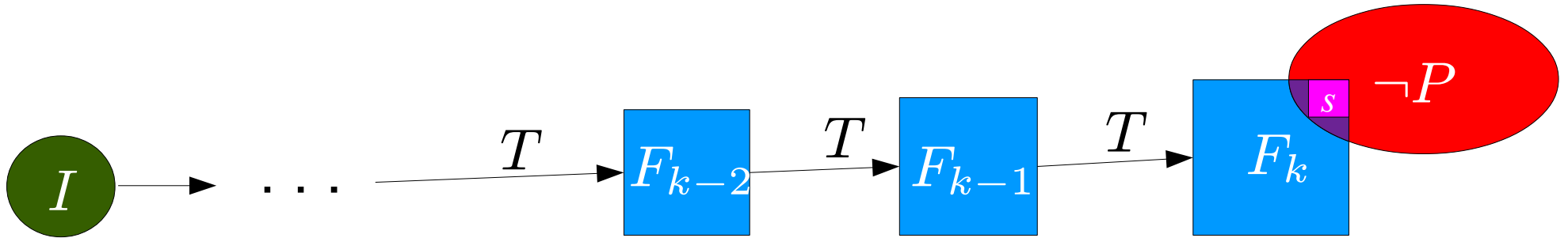
- Given a symbolic transition system and invariant property P , build an **inductive invariant** F s.t. $F \models P$
 - **Trace** of formulae $F_0(X) \equiv I, \dots, F_k(X)$ s.t:
 - for $i > 0$, F_i is a **set of clauses**
overapproximation of states reachable in up to i steps
- $F_{i+1} \subseteq F_i$ (so $F_i \models F_{i+1}$)
- $F_i \wedge T \models F'_{i+1}$
- for all $i < k$, $F_i \models P$

A (very) high level view of IC3



- **Blocking phase:** incrementally strengthen trace until $F_k \models P$
 - Get bad cube s
 - Call SAT solver on $F_{k-1} \wedge \neg s \wedge T \wedge s'$
(i.e., check if $F_{k-1} \wedge \neg s \wedge T \models \neg s'$)

A (very) high level view of IC3



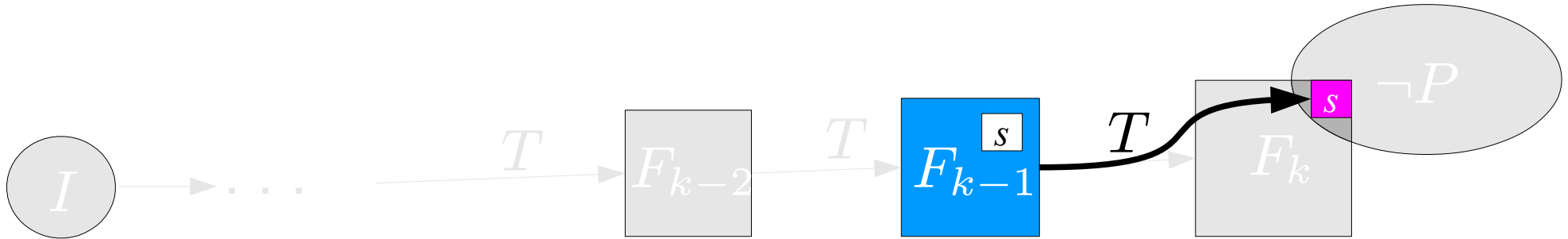
■ **Blocking phase:** incrementally strengthen trace until $F_k \models P$

■ Get bad cube s

■ Call SAT solver on $F_{k-1} \wedge \neg s \wedge T \wedge s'$
(i.e., check if $F_{k-1} \wedge \neg s \wedge T \models \neg s'$)

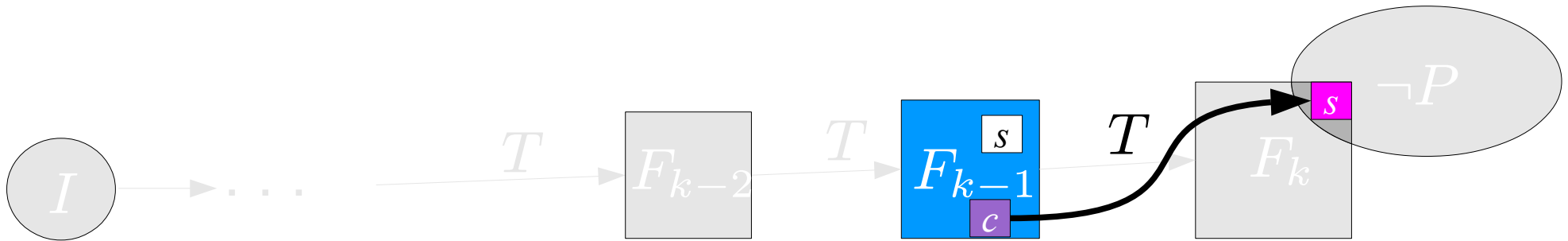
Check if s is inductive relative to F_{k-1}

A (very) high level view of IC3



- **Blocking phase:** incrementally strengthen trace until $F_k \models P$
 - Get **bad cube** s
 - Call SAT solver on $F_{k-1} \wedge \neg s \wedge T \wedge s'$
(i.e., check if $F_{k-1} \wedge \neg s \wedge T \models \neg s'$)

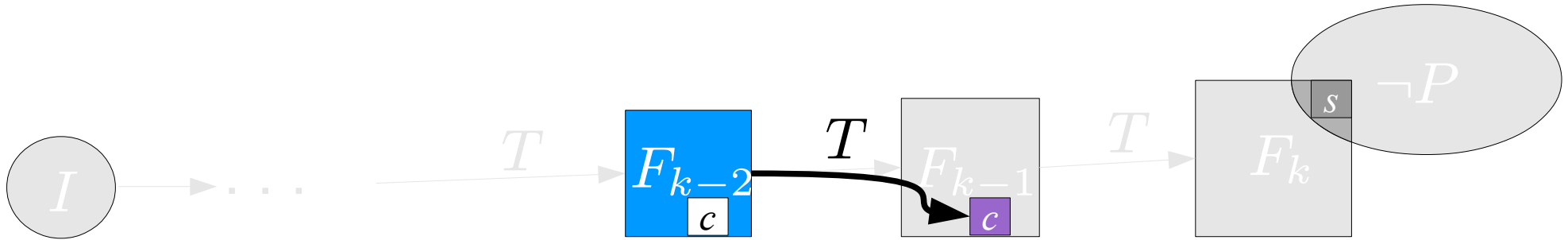
A (very) high level view of IC3



- **Blocking phase:** incrementally strengthen trace until $F_k \models P$
 - Get **bad cube** s
 - Call SAT solver on $F_{k-1} \wedge \neg s \wedge T \wedge s'$
 - **SAT:** s is reachable from $F_{k-1} \wedge \neg s$ in 1 step
 - Get a **cube** c in the preimage of s and try (recursively) to prove it unreachable from F_{k-2}, \dots
 - c is a **counterexample to induction** (CTI)

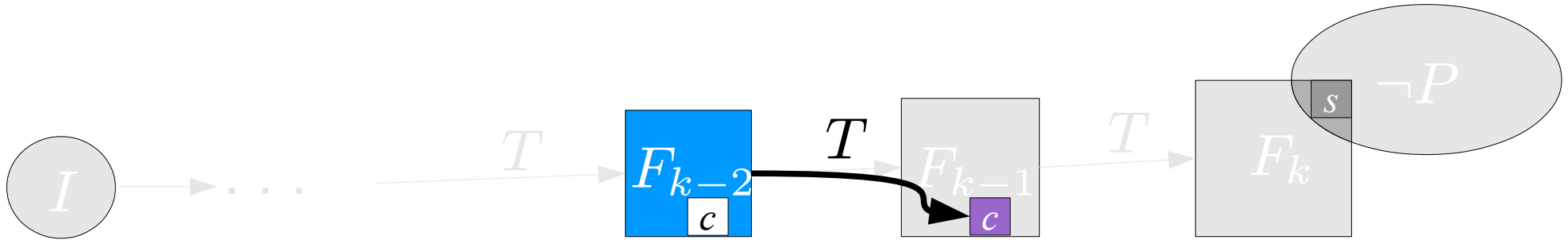
If I is reached,
counterexample
found

A (very) high level view of IC3



- **Blocking phase:** incrementally strengthen trace until $F_k \models P$
 - Get **bad cube** s
 - Call SAT solver on $F_{k-2} \wedge \neg s \wedge T \wedge s'$

A (very) high level view of IC3



■ **Blocking phase:** incrementally strengthen trace until $F_k \models P$

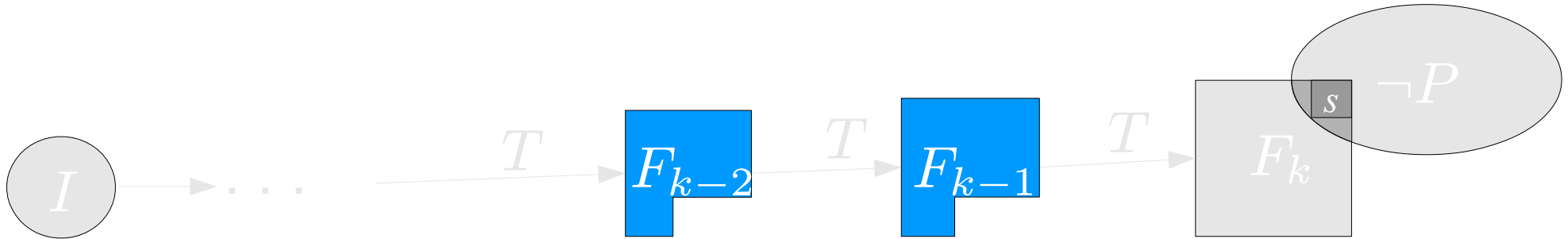
■ Get **bad cube** s

■ Call SAT solver on $F_{k-2} \wedge \neg s \wedge T \wedge s'$

■ **UNSAT:** $\neg c$ is **inductive relative** to F_{k-2} $F_{k-2} \wedge \neg c \wedge T \models \neg c'$

■ **Generalize** c to g and **block** by adding $\neg g$ to $F_{k-1}, F_{k-2}, \dots, F_1$

A (very) high level view of IC3



■ **Blocking phase:** incrementally strengthen trace until $F_k \models P$

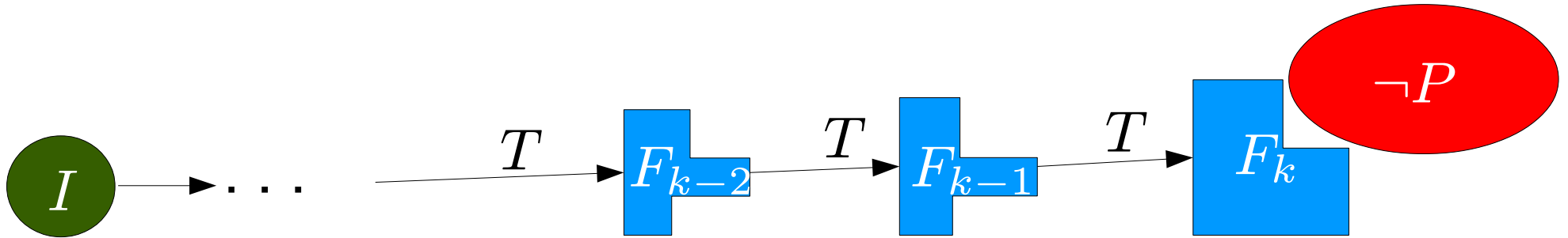
■ Get **bad cube** s

■ Call SAT solver on $F_{k-2} \wedge \neg s \wedge T \wedge s'$

■ **UNSAT:** $\neg c$ is **inductive relative** to F_{k-2} $F_{k-2} \wedge \neg c \wedge T \models \neg c'$

■ **Generalize** c to g and **block** by adding $\neg g$ to $F_{k-1}, F_{k-2}, \dots, F_1$

A (very) high level view of IC3



Propagation: extend trace to F_{k+1} and push forward clauses

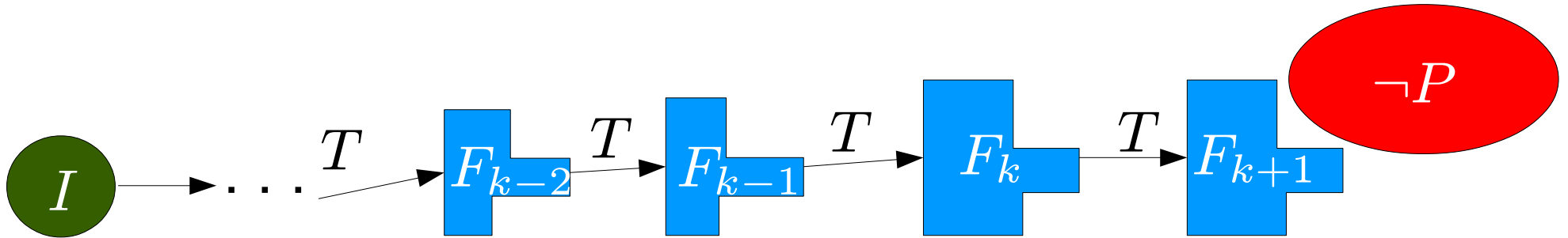
For each i and each clause $c \in F_i$:

Call SAT solver on $F_i \wedge T \wedge \neg c'$

If UNSAT, add c to F_{i+1}

$$F_i \wedge T \models c'$$

A (very) high level view of IC3



Propagation: extend trace to F_{k+1} and push forward clauses

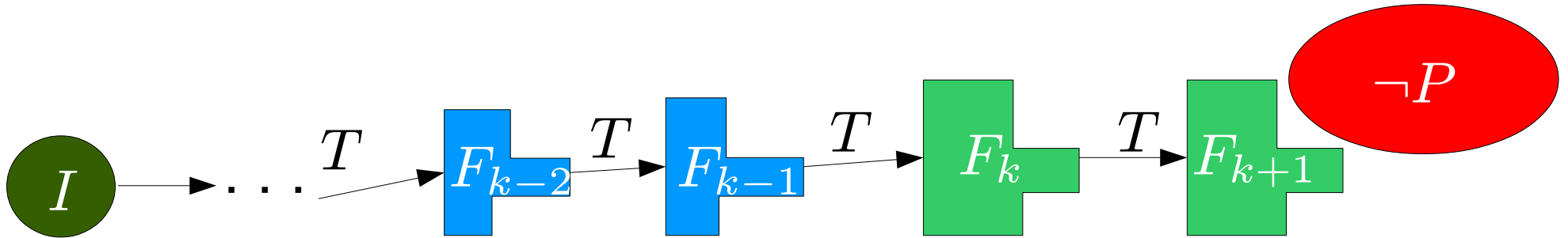
For each i and each clause $c \in F_i$:

Call SAT solver on $F_i \wedge T \wedge \neg c'$

If UNSAT, add c to F_{i+1}

$$F_i \wedge T \models c'$$

A (very) high level view of IC3



Propagation: extend trace to F_{k+1} and push forward clauses

For each i and each clause $c \in F_i$:

Call SAT solver on $F_i \wedge T \wedge \neg c'$

If UNSAT, add c to F_{i+1}

$$F_i \wedge T \models c'$$

If $F_i \equiv F_{i+1}$, P is proved,

otherwise start another round of blocking and propagation

- Crucial step of IC3

- Given a relatively inductive clause $c \stackrel{\text{def}}{=} \{l_1, \dots, l_n\}$
compute a **generalization** $g \subseteq c$ that is still inductive

$$F_{i-1} \wedge T \wedge g \models g' \quad (1)$$

- Drop literals from c and check that (1) still holds
 - Accelerate with unsat cores returned by the SAT solver
- 3 variants considered
 - **lattice-based** exploration of FMCAD'07 (original IC3)
 - Simple “**iterative**” algorithm (ABC, Tip)
 - **CTG-based** algorithm of FMCAD'13

Simple iterative generalization

```
void indgen(c, i):
    done = False
    for iter = 1 to max_iters:
        if done:
            break
        done = True
        for each l in c:
            cand = c \ {l}
            if not is_sat(I & cand) and
               not is_sat(trace[i] & ~cand & T & cand'):
                c = get_unsat_core(cand)
                done = False
                break
```

Lattice-based generalization

```
void indgen(c, i):
    fail = 0
    for each l in c:
        cand = c \ {l}
        if down(cand, i):
            c = up(cand, i)
            fail = 0
        elif ++fail > max_fail:
            break

bool down(c, i):
    while True:
        if is_sat(I & c): return False
        if not is_sat(trace[i] & ~c & T & c'):
            c = get_unsat_core(c)
            return True
        else:
            s = get_predecessor(i, T, c')
            c = lattice_join(c, s)
```


CTG-based generalization

```
void indgen(c, i, d=0):  
    f = 0  
    for each l in c:  
        cand = c \ {l}  
        if ctg_down(cand,  
                    i, d):  
            c = cand  
    elif ++f > max_f:  
        break
```

```
bool ctg_down(c, i, rec_depth):  
    ctgs = 0  
    while True:  
        if is_sat(I & c): return False  
        if not is_sat(trace[i] & ~c & T & c'):  
            c = get_unsat_core(c)  
            return True  
        elif rec_depth > max_depth: return False  
        else:  
            s = get_predecessor(i, T, c')  
            if ctgs < max_ctgs and i > 0 and  
                not is_sat(I & s) and  
                not is_sat(trace[i-1] & ~s & T & s'):  
                ctgs += 1  
                j = find_max_level(i, s)  
                indgen(s, j-1, rec_depth+1)  
                trace[j].append(s)  
            else:  
                ctgs = 0  
                c = lattice_join(c, s)
```

An important optimization

```
void indgen(c, i):  
    f = 0  
    for each l in c:  
        cand = c \ {l}  
        if down(cand, i):  
            c = up(cand, i)  
    elif ++f > max_f:  
        break
```

```
bool down(c, i):  
    while True:  
        if is_sat(I & c):  
            return False  
        if not is_sat(trace[i] & ~c & T & c'):  
            c = get_unsat_core(c)  
            return True  
    else:  
        s = get_predecessor(i, T, c')  
        cj = lattice_join(c, s)  
        c = cj
```

An important optimization

```
void indgen(c, i):  
    required = {}  
    f = 0  
    for each l in c:  
        cand = c \ {l}  
        if down(cand, i):  
            c = up(cand, i)  
        else:  
            if ++f > max_f:  
                break  
            required.add(1)
```

```
bool down(c, i):  
    while True:  
        if is_sat(I & c):  
            return False  
        if not is_sat(trace[i] & ~c & T & c'):  
            c = get_unsat_core(c)  
            return True  
    else:  
        s = get_predecessor(i, T, c',  
                             no_lifting)  
        cj = lattice_join(c, s)  
        if (c \ cj) ∩ required != {}:  
            return False  
        c = cj
```

Applies to CTGs as well

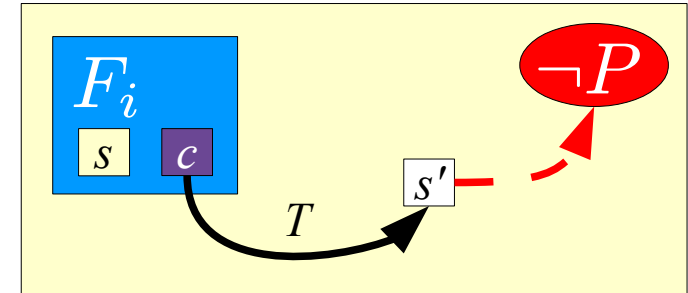
[thanks to A. Bradley]

- When $F_i \wedge \neg s \wedge T \wedge s'$ is satisfiable:

- s reaches $\neg P$ in $k-i$ steps

- s can be reached from F_i in 1 step

- strengthen F_i by blocking cubes c in the preimage of s



- Extract CTI c from the SAT assignment

- And generalize (“lift”) to represent multiple bad predecessors

- 2 variants considered

- Simple syntactic generalization based on cone-of-influence (similar to original IC3)

- SAT-based generalization with unsat cores of FMCAD'11

- (effects similar to ternary simulation)

SAT-based CTI generalization

```
void lift(cti, inputs, next):  
  for i = 1 to max_iters:  
    b = is_sat(cti & inputs & T & ~next')  
    assert not b # assume T to be functional  
    c = get_unsat_core(cti)  
    if should_stop(c, cti):  
      break  
  cti = c
```

Other high-level aspects

[See the paper for details]

- Management of proof obligations of cubes to block (c, i)
 - recursive (i.e. **stack**-based) procedure vs **priority queue**
- Target enlargement
 - **0-step** (PDR FMCAD'11), **1-step** (original IC3), **4-step** (Tip)
- Model preprocessing with sequential simplifications
 - no preprocessing vs 2-step temporal decomposition and detection of equivalences

- A number of “low-level” settings can significantly affect the performance of IC3. **We considered:**
 - **SAT solver:** minisat-core, minisat-simp, picosat
 - **CNF conversion:** standard Tseitin-like, “ABC-like” [SAT'07]
 - **# of SAT solver instances:** single instance, one per frame
 - **Activity of literals** for inductive generalization: on, off
 - **Approximated SAT queries** in inductive generalization
 - **Bound on the number of decisions:** if reached, report SAT
 - To reduce the runtime on satisfiable instances
 - Only applied during inductive generalization, does not affect correctness nor completeness
 - Use a static bound of 100 decisions in the experiments

Setup of the experiments

- **Benchmarks:** single track of HWMCC '11, '12, '13
 - Include all instances, even the “questionable” ones (e.g. bwd beam)
- **Machine:** 2.5Ghz Xeon X5650 CPU, 96Gb RAM, 12Mb cache, 900s time limit, 4Gb mem limit
- **Baseline configuration** following the PDR description of [FMCAD'11]
- **Other configurations change only a single parameter**
 - Assumption of independence of parameters
 - Necessary simplification to avoid combinatorial explosion
- Implementation available at <https://nuxmv.fbk.eu/tests/difts2014>

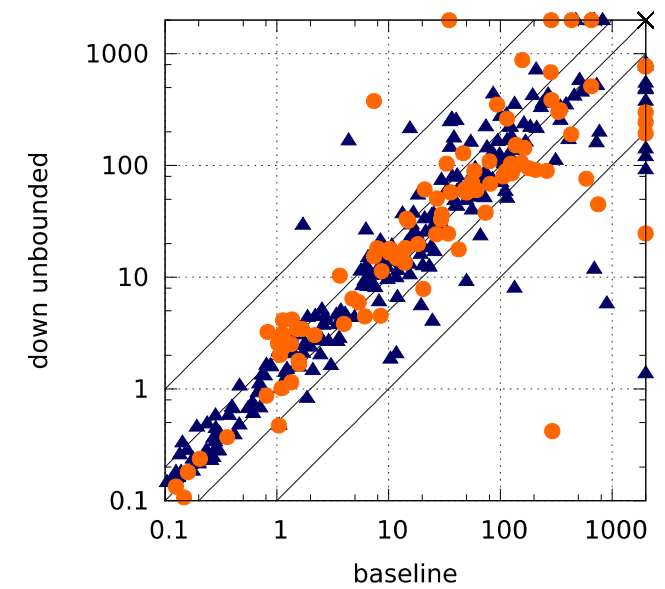
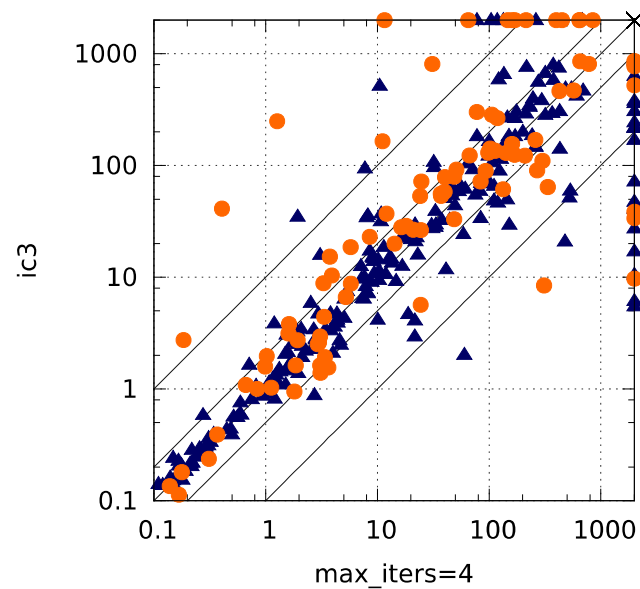
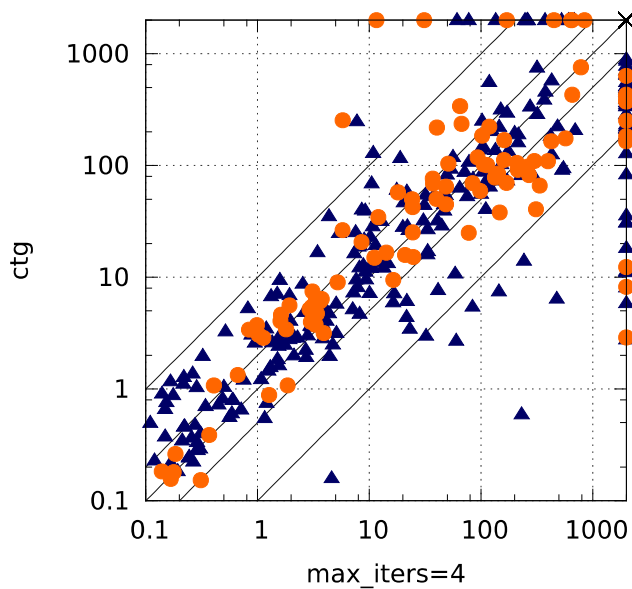
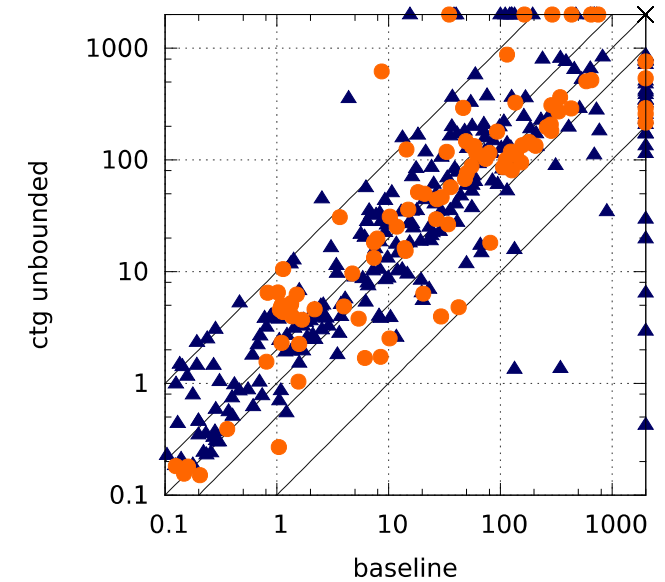
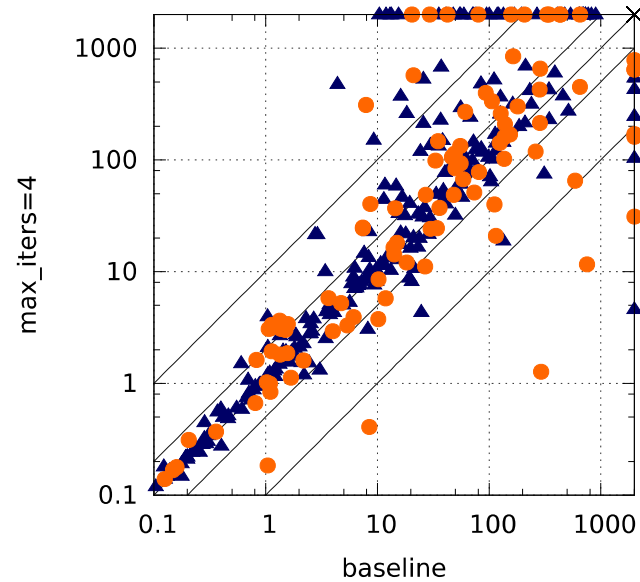
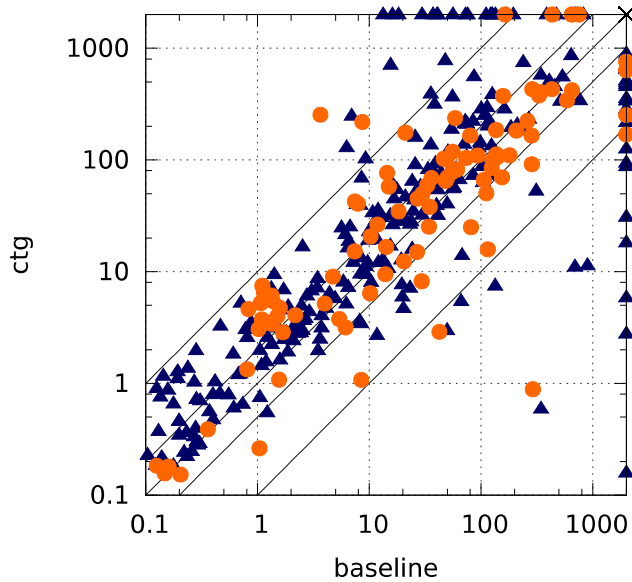
- Configurations tested in the paper:
 - **baseline**: iterative algorithm with `max_iters=+∞`, “round robin” loop over literals to drop
 - Essentially, the **Tip** way
 - **indgen-ic3**: lattice-based generalization with `max_fails=3`, apply **up** only for cubes with `>25` literals
 - The strategy of the original **IC3** paper
 - **indgen-ctg**: ctg-based generalization with `max_fails=3`, never apply **up**, use `max_ctgs=3`, `max_depth=1`
 - What is described in **FMCAD'13**
 - **no relative induction**: only check $F_i \wedge T \models c'$

- Further configurations tested (not in the paper):
 - `indgen-ctg` and `indgen-ic3` with the `optimization` described earlier
 - `iterative` with `max_iters=4` and `max_iters=64`
 - `down unbounded`: lattice-based generalization with `max_fails=+∞`, `no up`
 - `ctg unbounded`: ctg-based generalization with `max_fails=+∞`

Results: inductive generalization

Configuration	# solved	Δ baseline	Gained	Lost	Total time
ctg unbounded	431	10	31	21	36681
down unbounded	427	6	16	10	28931
iterative max_iters=64	422	1	10	9	29985
baseline (iter unbounded)	421	0	0	0	27242
ctg (optimized)	417	-4	25	29	31736
ctg (naive)	412	-9	26	35	34632
ic3 (optimized)	396	-25	9	36	30388
iterative max_iters=4	386	-35	11	46	25541
ic3 (naive)	384	-37	7	44	30405
no relative induction	381	-40	5	45	32539

Inductive generalization

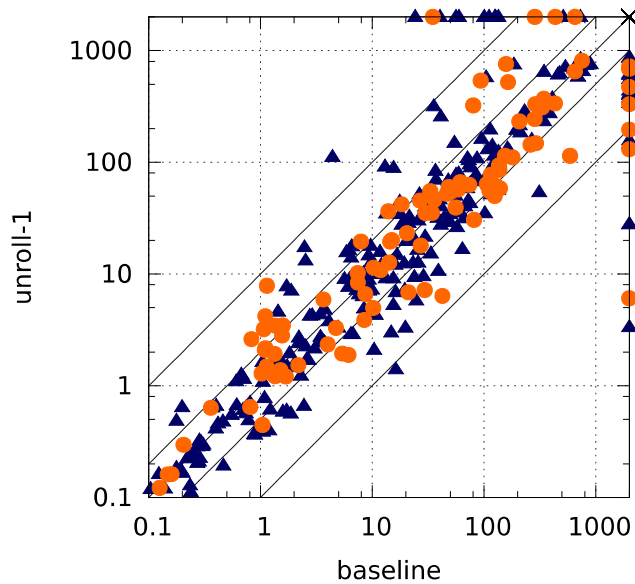


Results: other high-level parameters

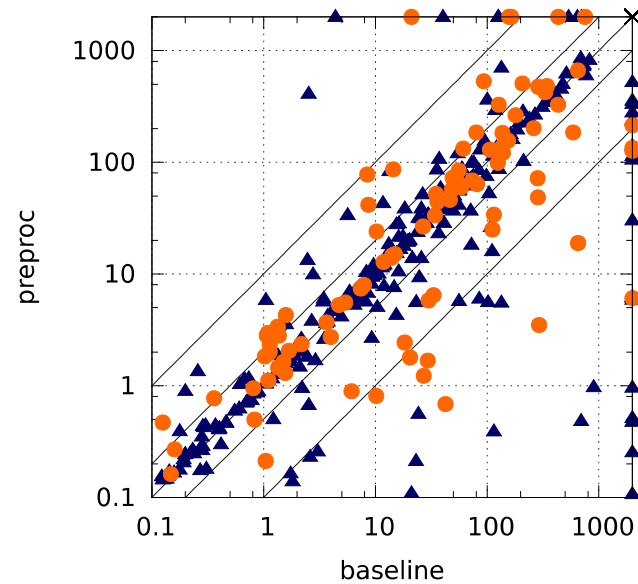
Configuration	# solved	Δ baseline	Gained	Lost	Total time
preprocessing	431	10	22	12	25399
unroll-4	431	10	29	19	31525
unroll-1	428	7	23	16	34005
baseline	421	0	0	0	27242
stack	401	-20	14	34	34447
syntactic CTI lifting	334	-87	9	96	31360

Other high-level parameters

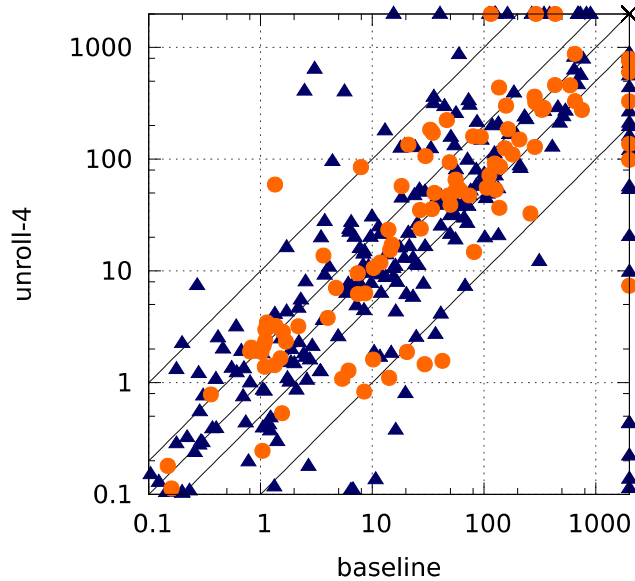
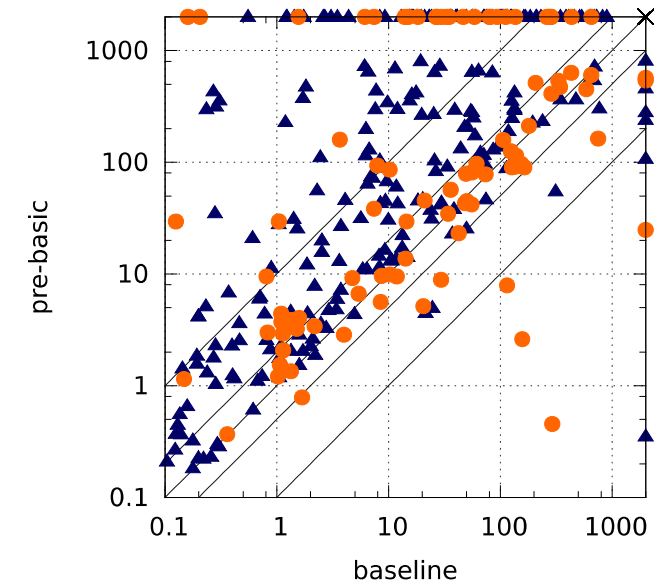
Target enlargement



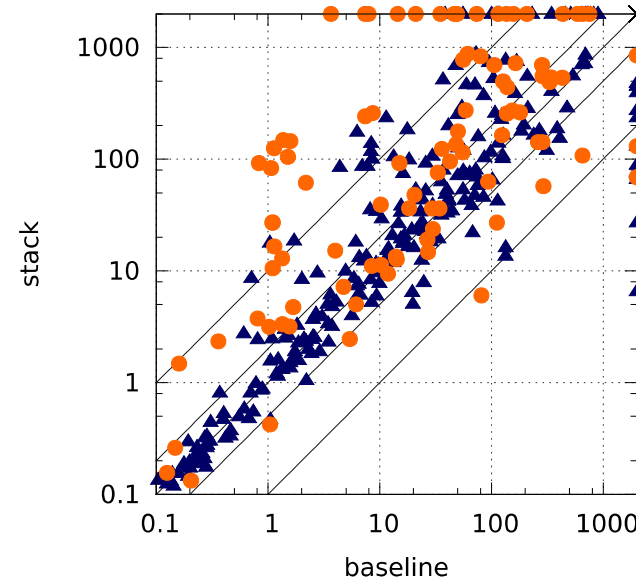
Preprocessing



CTI lifting



Queue management

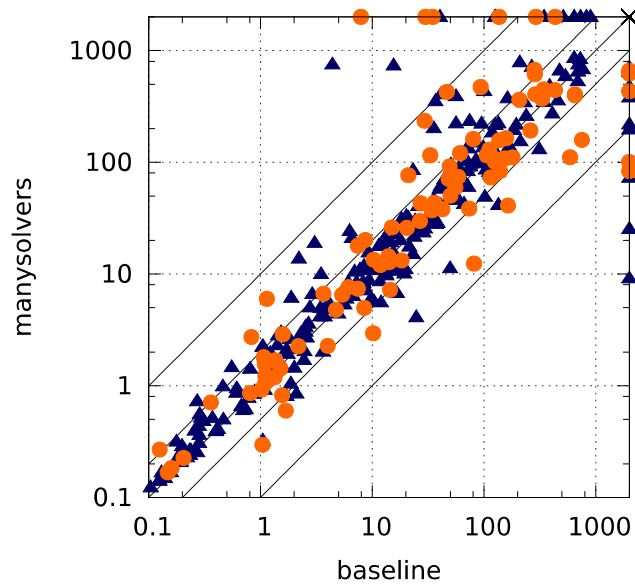


Results: low-level parameters

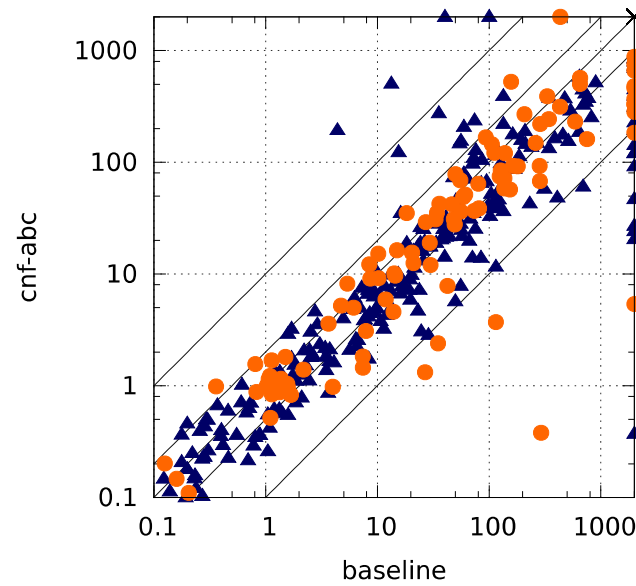
Configuration	# solved	Δ baseline	Gained	Lost	Total time
cnf-abc	448	27	30	3	28431
sat-approx	438	17	26	9	28833
activity	430	9	17	8	31352
minisat-simp	429	8	18	10	27413
baseline	421	0	0	0	27242
manysolvers	420	-1	14	15	32306
picosat	402	-19	11	30	32570

Low-level parameters

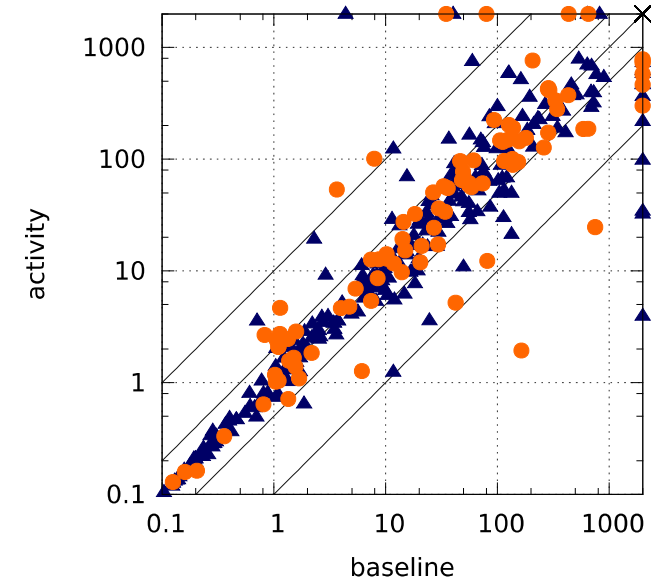
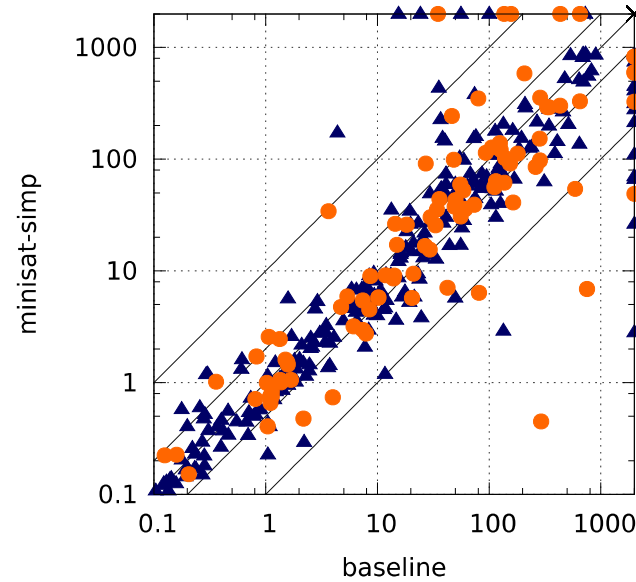
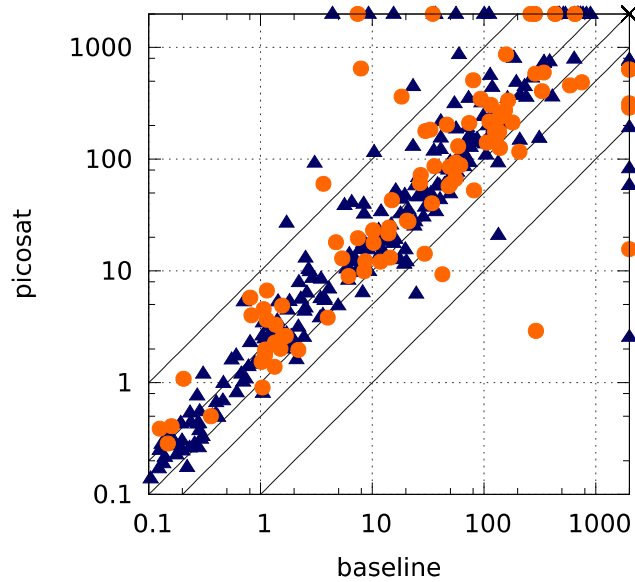
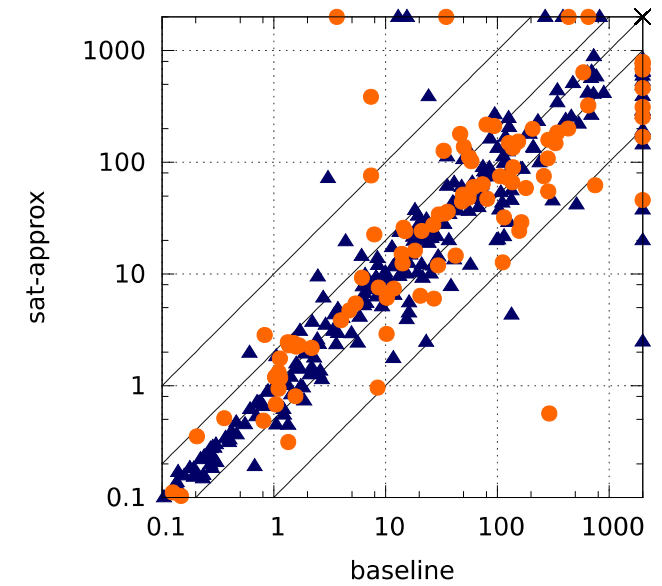
SAT solver



CNF conversion



Indigen tweaks

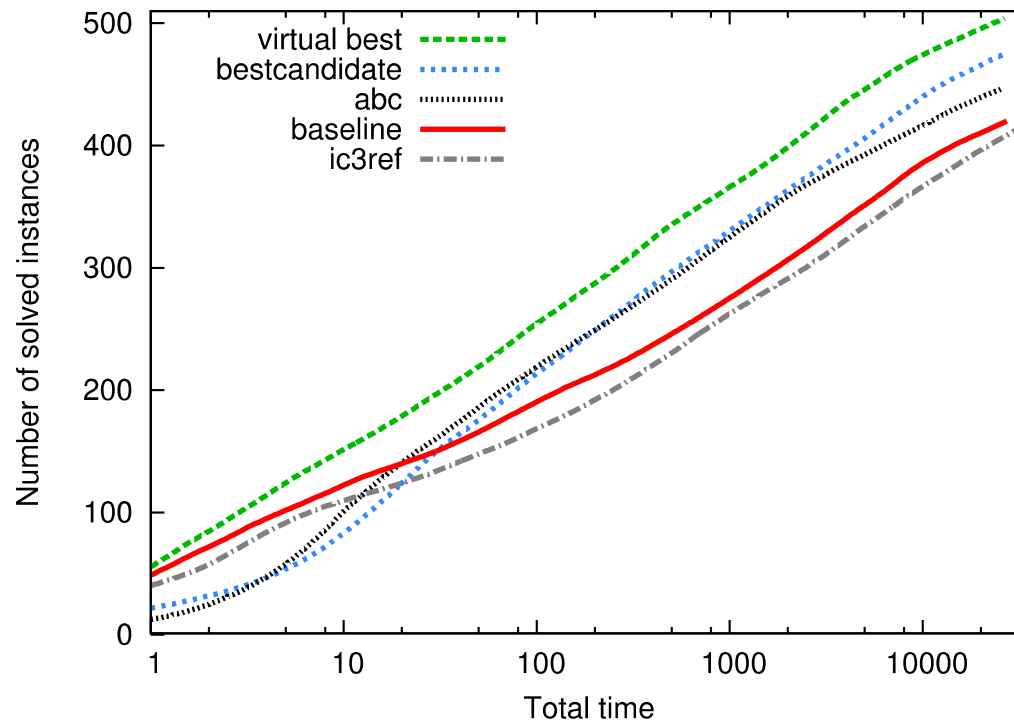


Comparison with the state of the art

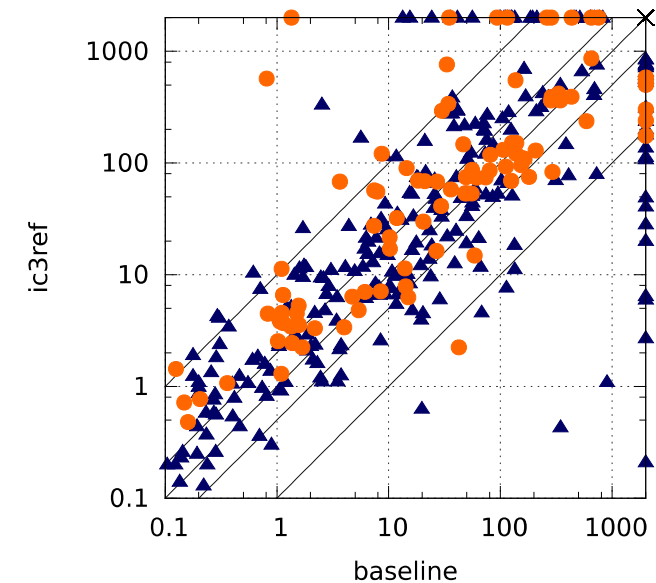
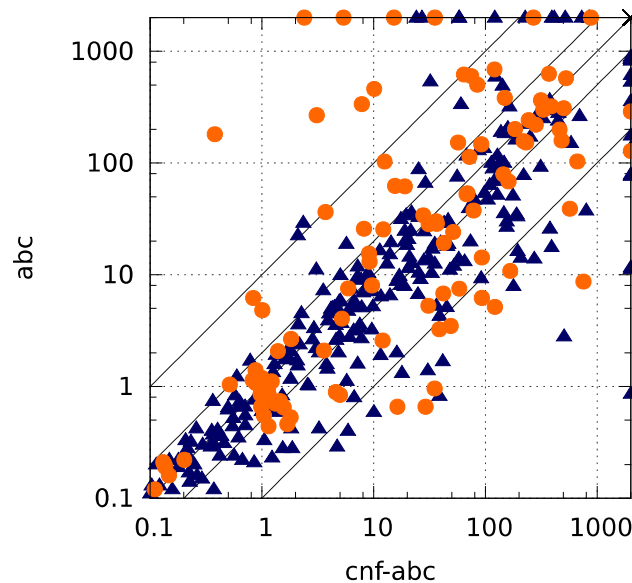
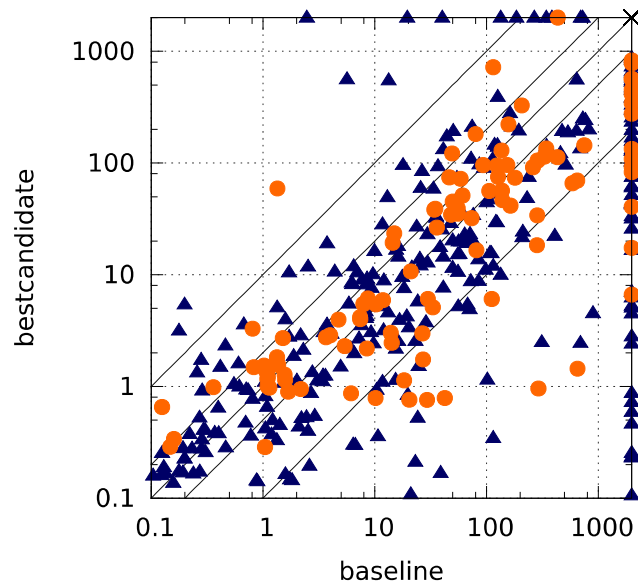
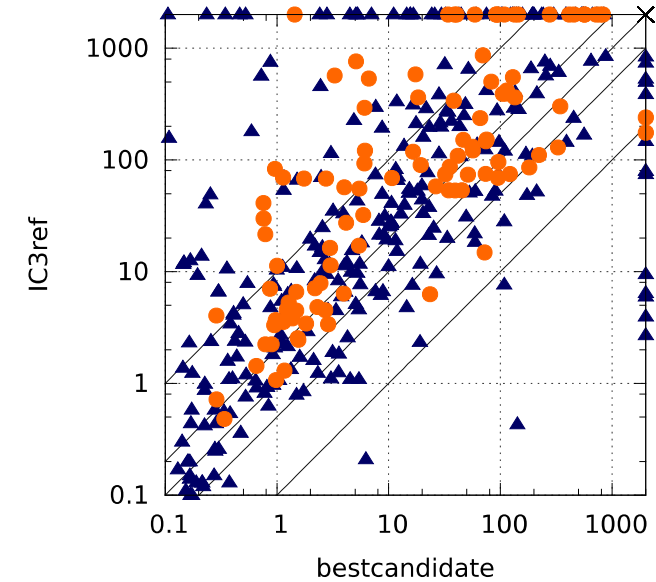
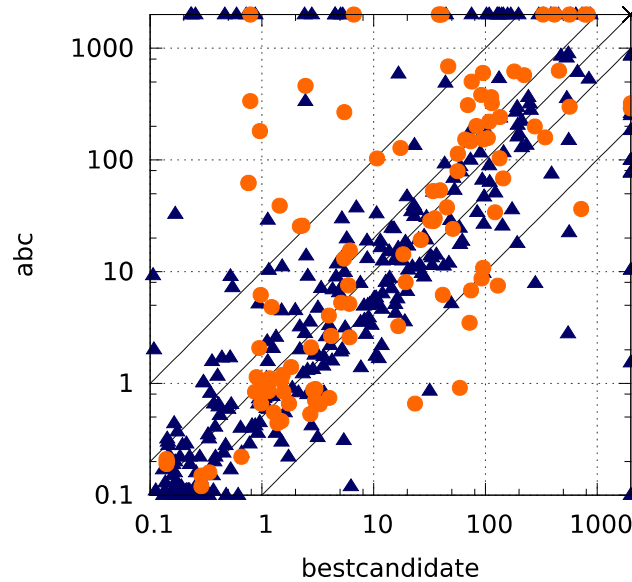
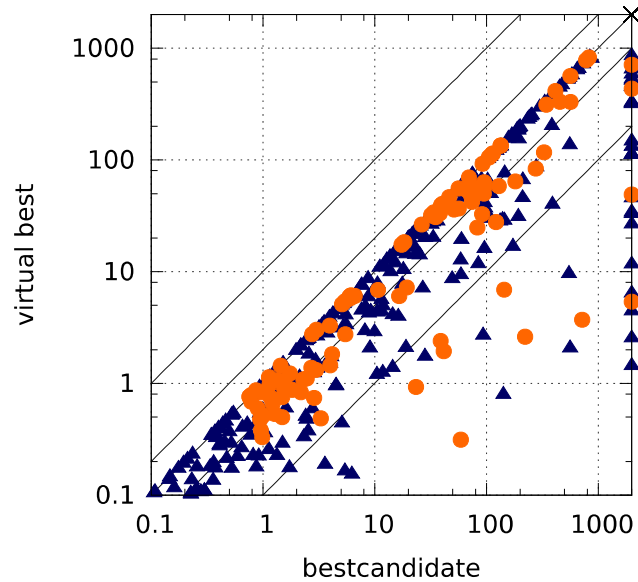
- **Bestcandidate** configuration (combination of all the improvements). **Baseline +**:
 - preproc, unroll-4, cnf-abc, activity, sat-approx, minisat-simp
- PDR implementation in **ABC**
- A. Bradley **IC3 reference** implementation
- **Virtual best** among all (our) configurations

Comparison with state of the art

Configuration	# solved	Δ baseline	Gained	Lost	Total time
virtual best	505	84	84	0	26509
bestcandidate	477	56	68	12	27341
ABC (PDR only)	447	26	37	11	25360
baseline	421	0	0	0	27242
IC3ref	418	-3	32	35	33924



Comparison with state of the art



- Systematic evaluation of IC3 variants/optimizations
 - Same implementation allows for better comparison
- Despite **limitations** (independence, only runtime compared), interesting results
 - Independent confirmation of results from the literature
 - Importance of some low-level parameters (**CNF conversion**)
- Future work
 - Improve the analysis of results (e.g. analyze other statistics, evaluate the dependency among parameters)
 - Consider further parameters (e.g. effect of recycling of SAT solvers)
 - Include recent IC3-like variants (e.g. Lazy Abstraction, Avy)

Thank You

IC3 pseudo-code

```
bool IC3(I, T, P):
    trace = [I]    # first elem of trace is init formula
    trace.push()   # add a new frame
    while True:
        # blocking phase
        while is_sat(trace.last() & ~P):
            c = extract_cube() # c |= trace.last() & ~P
            if not rec_block(c, trace.size()-1):
                return False # counterexample found

        # propagation phase
        trace.push()
        for i=1 to trace.size()-1:
            for each cube c in trace[i]:
                if not is_sat(trace[i] & ~c & T & c'):
                    trace[i+1].append(c)
        if trace[i] == trace[i+1]:
            return True # property proved
```

IC3 pseudo-code

```
bool rec_block(s, i):  
    if i == 0:  
        return False # reached initial states  
    while is_sat(trace[i-1] & ~s & T & s'):  
        c = get_predecessor(i-1, T, s')  
        if not rec_block(c, i-1):  
            return False  
    g = generalize(~s, i)  
    trace[i].append(g)  
    return True
```

IC3 pseudo-code

```
bool rec_block(s, i):  
    if i == 0:  
        return False # reached initial states  
    while is_sat(trace[i-1] & ~s & T & s'):  
        c = get_predecessor(i-1, T, s')  
        if not rec_block(c, i-1):  
            return False  
        g = generalize(~s, i)  
        trace[i].append(g)  
    return True
```

Predecessor computation

Inductive generalization

Proof obligation management

- A **proof obligation** (c, i) represents a cube reaching the bad states in $k-i$ steps (k : number of frames in the trace)
- c is **bad regardless** of the value of i
 - if later another obligation (c, j) , $j > i$ is generated, c must be blocked again
 - when (c, i) is blocked, **try blocking also** $(c, i+1)$
 - from stack to **priority queue** (ordered by i)
- **2 variants considered**
 - simple recursive procedure (**stack** based)
 - **priority queue**

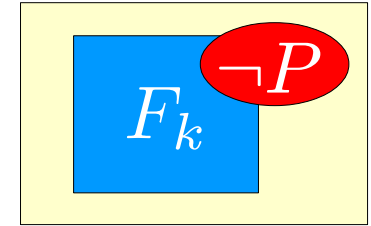
Queue-based blocking

```
bool rec_block(s, i):
    q = PriorityQueue()
    q.push((s, i))
    while not q.empty():
        (c, j) = q.top()
        if j == 0:
            return False # reached initial states
        if is_sat(trace[j-1] & ~c & T & c'):
            p = get_predecessor(j-1, T, c')
            q.push((p, j-1))
        else:
            q.pop()
            g = generalize(~c, j)
            trace[j].append(g)
            if j < trace.size(): # try blocking at later frames
                q.push((c, j))
    return True
```

Target enlargement

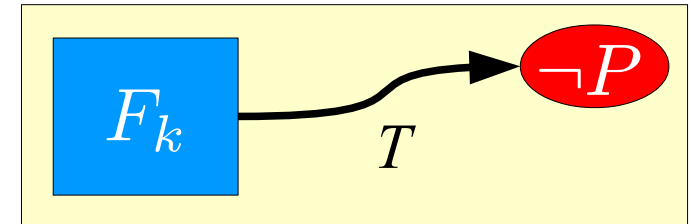
- The IC3 presented here is the **PDR variant** [ABC FMCAD'11]

- for all $i < k$, $F_i \models P$
- Blocking starts from $F_k \cap \neg P$



- **Original IC3:**

- $F_i \models P$ for all i (including F_k)
- Blocking starts from $F_k \wedge T \models P'$



- PDR can emulate IC3 with 1-step **target-enlargement**

- Preprocess the model replacing P with $\text{unroll}(1, P)$

- **3 variants considered**

- **no** enlargement (PDR-like)
- **1-step** enlargement (IC3-like)
- **4-step** enlargement (Tip-like)

- **Preprocessing** with sequential simplification techniques often critical for performance
 - E.g. winners of HWMCC
- **2 variants considered**
 - **no preprocessing**, just IC3
 - 2-step **temporal decomposition** + detection of **equivalences** (with ternary simulation)