Trading-off incrementality and dynamic restarts of multiple solvers in IC3

Marco Palena
Formal Methods Group
Politecnico di Torino
marco.palena@polito.it

Outline

- Preliminaries
- IC3 algorithm
- Characterization of SAT solving in IC3
- Incremental loading of transition relation
- SAT solvers clean-up heuristics
- Conclusions and future works

Outline

- **Preliminaries**
- IC3 algorithm
- Characterization of SAT solving in IC3
- Incremental loading of transition relation
- SAT solvers clean-up heuristics
- Conclusions and future works

Preliminaries

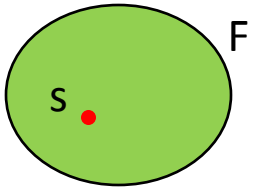
- **Context:** unbounded model checking for hardware verification
- Boolean circuits modelled as **finite state transition systems**:

$$S = \langle \boldsymbol{x}, I, T \rangle$$

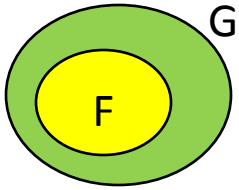
  - $\boldsymbol{x} = \{x_0, x_1, ..., x_n\}$: **state variables**.
  - $I(\boldsymbol{x})$: **initial states**.
  - $T(\boldsymbol{x}, \boldsymbol{x}')$: **transition relation**.
- **State** = complete assignments to the state variables.
  - Primed variables denotes future states
- **Boolean formulas**: represent set of states
  - **Literal** is a state variable or its negation: e.g. x_1 , $\neg x_2$
  - **Cube** is a conjunction of literals: e.g. $x_1 \wedge \neg x_2 \wedge x_3$
  - **Clause** is a disjunction of literals: e.g. $\neg x_1 \vee x_2 \vee x_3$
  - **CNF** is a conjunction of clauses: e.g. $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee x_3)$

Preliminaries

- An assignment s **satisfies** F if F evaluates to **true** under s: $s \models F$

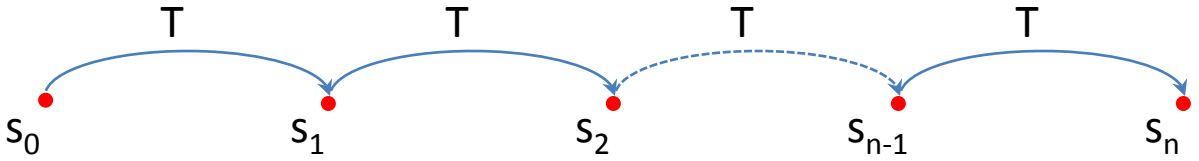


- F is **stronger** than G if: $F \rightarrow G$



- When an assignment (s, t') satisfies T:
  - s is a **predecessor** of t
  - t is a **successor** of s

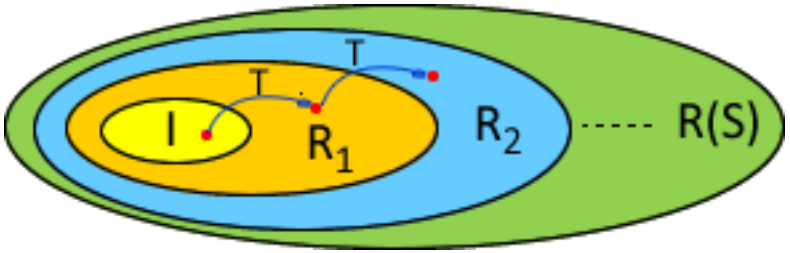
- $s_0, s_1, ..., s_n$ is a **path** if $s_i, s_{i+1} \models T \quad \forall i : 0 \leq i < n$



Preliminaries

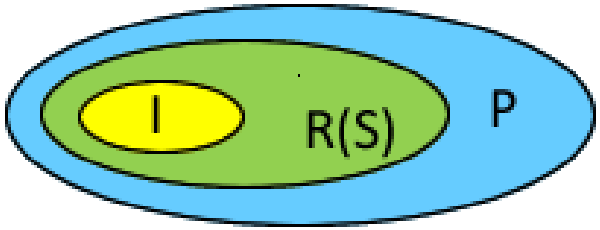
- A state s is **reachable** if there exists a path $s_0, s_1, ..., s_n = t : s_0 \models I$
  - Set of **n-bounded reachable states** of S: $R_n(S)$
  - Set of **reachable states** of S:

$$R_n(S) = \bigcup_{i \geq 0} R_n(S)$$



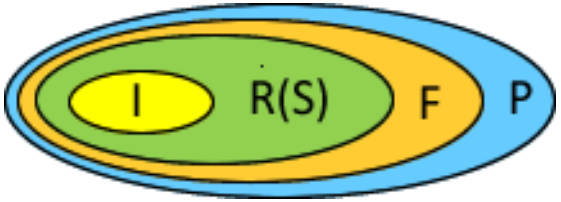
- Given $S = \langle \boldsymbol{x}, I, T \rangle$ and **property** $P(\boldsymbol{x})$ the **invariant verification problem** (IVP) is:

$$\forall s \in R(S) : s \models P \ ?$$



Preliminaries

- F is an **inductive invariant** of S:
  - Base case: $I \rightarrow F$
  - Inductive case: $F \wedge T \rightarrow F'$
- F is an **inductive invariant relative to** G:
  - Base case: $I \rightarrow F$
  - Inductive case: $G \wedge F \wedge T \rightarrow F'$
- An inductive invariant P is an over-approximation to reachable states $\Longrightarrow$ IVP can be seen as the problem to find an inductive invariant F stronger than the property P (**inductive strengthening** of P): $F \rightarrow P$



Outline

- Preliminaries
- **IC3 algorithm**
- Characterization of SAT solving in IC3
- Incremental loading of transition relation
- SAT solvers clean-up strategies
- Conclusions and future works

IC3

- Incremental SAT-based invariant verification algorithm that uses induction
- Maintains a set of over-approximations to bounded reachable states (**time frames**):

$$\boldsymbol{F_k} = F_0, F_1, ..., F_k$$

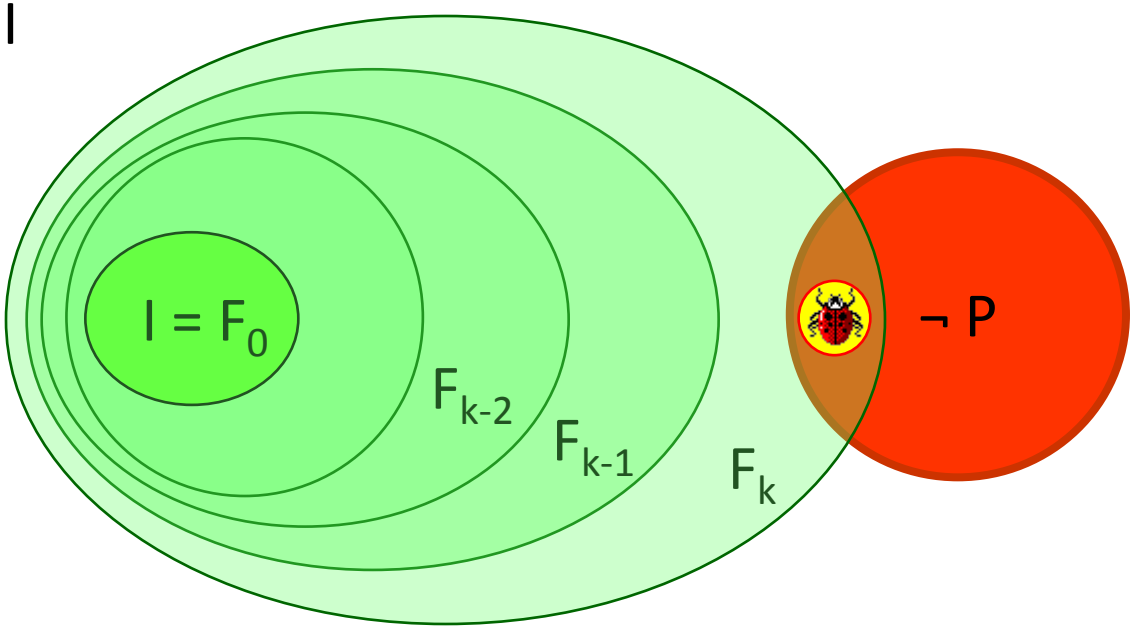
  - Tries to find an inductive strengthening of P incrementally refining $\boldsymbol{F_k}$ with relative inductive clauses
- The following conditions hold throughout the algorithm:
  - $F_0 = I$ **(C1)**
  - For each $1 \leq i < k$:
    - $F_{i+1} \rightarrow F_i$ **(C2)**
    - $F_i \wedge T \rightarrow F'_{i+1}$ **(C3)**
    - $F_i \rightarrow P$ **(C4)**

IC3 algorithm

- At iteration k, IC3 enumerates states of F_k that violate P:

$$SAT?[F_k \wedge \neg P] \quad \textbf{(Q1)}$$

- Extends the bad state found into a bad cube
- Every state (or cube) that can reach violation of P discovered for F_k must be **blocked** i.e. proved unreachable within k steps from I

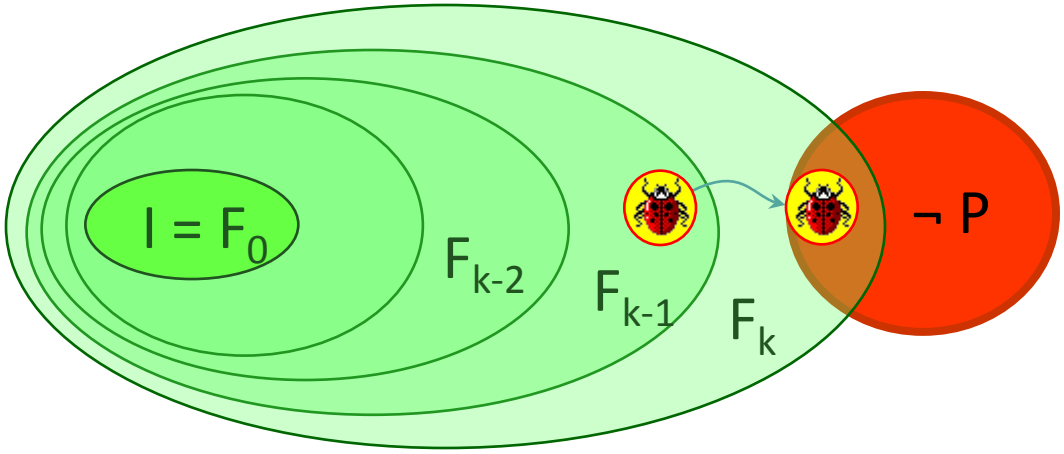


Cube blocking procedure

- To block a cube *c* in F_k , IC3 first tries to find out if ¬*c* is inductive relative to F_{k-1} :

$$\text{SAT?}[F_{k-1} \wedge \neg c \wedge T \wedge c'] \qquad \textbf{(Q2)}$$

- If not, a predecessor *s* is discovered $\Longrightarrow$ s must be blocked in in F_{k-1} first $\Longrightarrow$ blocking of c is delayed, the procedure tries to block *s* in F_{k-1} $\Longrightarrow$ blocking procedure iterates
  - Eventually either ¬*c* become inductive relative to F_{k-1} or a predecessor in F_0 is found (path from initial states to a bad cube)



Cube blocking procedure

- If (Q2) is UNSAT, a clause ¬*c* that is inductive relative to F_{k-1} is found, then IC3 tries to remove literals from ¬*c* to obtain an **inductive generalization**
  - Removing literals can break relative induction!
- For each literal removed, relative induction must be checked again:
  - **Inductive case**: SAT?[F_{k-1} ∧ cls ∧ T ∧ ¬cls'] **(Q4)**
  - **Base case**: SAT?[I ∧ ¬cls] **(Q5)**
- A delayed cube can become blocked as a result of the blocking of a deeper cube:
  - When the blocking of a delayed cube is resumed, IC3 checks if it still needs to be blocked:

SAT?[F_k∧ c] **(Q3)**

Propagation phase

- When every bad state in F_k has been enumerated and blocked, IC3 instantiates a new time frame and tries to propagate each clause in $F_i : 1 \leq i < k$ forward on F_{i+1}

  SAT?[$F_k \wedge T \wedge \neg cls'$] **(Q6)**

  - If SAT, the clause cls is added to F_{k+1}
- If during the propagation phase is discovered that $F_i \equiv F_{i+1}$ for some $1 \leq i < k \implies F_i$ is an inductive strengthening for P

Outline

- Preliminaries
- IC3 algorithm
- **Characterization of SAT solving in IC3**
- Incremental loading of transition relation
- SAT solvers clean-up heuristics
- Conclusions and future works

SAT solvers use in IC3

- IC3 is a SAT-based invariant verification algorithm
  - Each SAT call has a small size compared to other SAT-based verification algorithms (no TR unrolling).
  - Huge amount of SAT calls ($\approx 10^3 - 10^6$)
- How to organize the underlying SAT solving work required?
  - **SAT solvers allocation strategies**
  - **SAT solvers loading strategies**
  - **SAT solvers clean-up strategies**
- Our implementation adopts a multiple solver approach (one solver for each time frame)

SAT queries breakdown in IC3

- **Types of queries:**
  - **(Q1) - Target intersection checks**: SAT?[$F_i \wedge \neg P$]
  - **(Q2) - Relative inductive check**: SAT?[$F_i \wedge \neg cube \wedge T \wedge cube'$]
  - **(Q3) - Blocked cube checks**: SAT?[$F_i \wedge cube$]
  - **(Q4) - Inductive generalization check**: SAT?[$F_i \wedge cls \wedge T \wedge \neg cls'$]
  - **(Q5) - Base of induction check**: SAT?[$I \wedge \neg cls$]
  - **(Q6) - Clause propagation check**: SAT?[$F_i \wedge T \wedge \neg cls'$]
- **HWMCC 2012** (time limit 900s, memory limit 2 GB): 70 solved instances/310

SAT call type	% calls	Num calls	Solving time
Target intersection	0.1%	483	81 ms
Relative induction	7.6%	31172	334 ms
Blocked cube	6.8%	27891	219 ms
Generalize	34.7%	142327	575 ms
Induction base	35.9%	147248	112 ms
Propagation	14.9%	61114	681 ms

Incremental SAT interface

- CNF subject to SAT queries vary widely from call to call:
  - Transition relation not always needed
  - Some queries assume a next state cube
- IC3 needs an **incremental SAT interface**
  - New clauses must be added
  - Clauses from previous calls must be removed
  - Literal assumptions must be made
- To remove clauses from the solver, **activation literals** are used:
  - **Deactivated clauses slow down SAT solving!**
    - $\Longrightarrow$ **Load as less clauses as needed**
    - $\Longrightarrow$ **Clean-up periodically each solver**

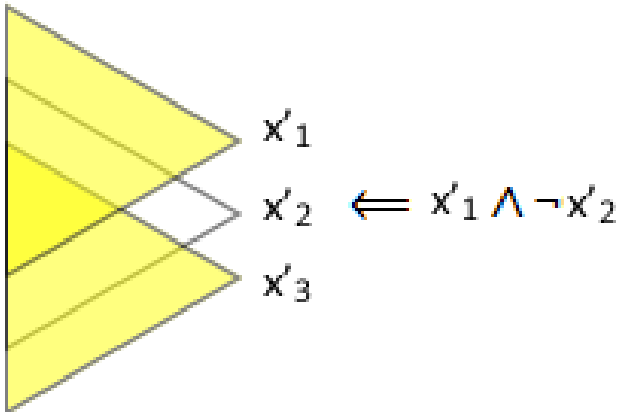
Outline

- Preliminaries
- IC3 algorithm
- Characterization of SAT solving in IC3
- **Incremental loading of transition relation**
- SAT solvers clean-up heuristics
- Conclusions and future works

Incremental loading of TR

- No need to load the whole TR in each solver [Een,Mishchenko,Brayton 2011]
  - Not every SAT call needs it
  - Every SAT call that needs it, also makes a literal assumption on next state

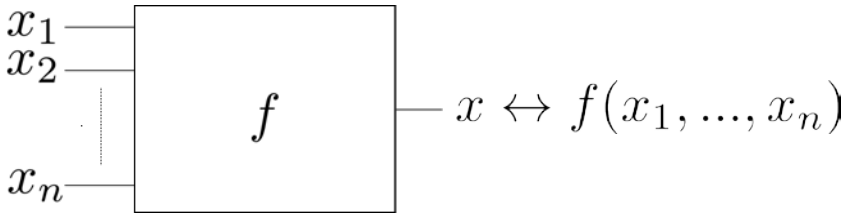
  $\Longrightarrow$ **Load just the transitive fanin (logic cone) of each variable in the next state cube assumed in the query**

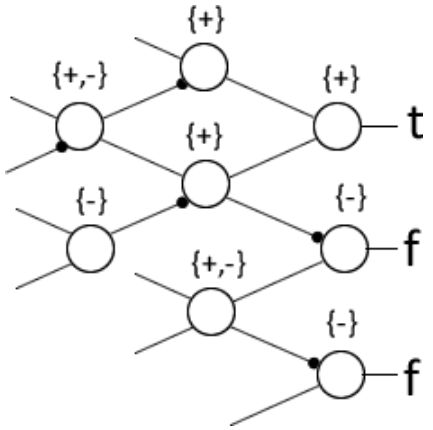


- Proved to be very effective!
- Problem: logic cones loaded from previous queries accumulate in each solver!

Plaisted-Greenbaum CNF encoding

- Each SAT query that needs TR, constraints next states with a cube c' $\Longrightarrow$ underlying TR's AIG is a constrained boolean circuit
- **Plaisted-Greenbaum encoding (PG)**:
  - Translates a constrained boolean circuit into a minimal set of clauses using **gate polarities**: {+} or {-}
  - Introduces for each gate an auxiliary variable x logically linked to its boolean function by means of a bi-implication





- Equisatisfiable CNF can be found translating just the left side of the bi-implication for {-} gates and/or the right side for {+} gates

Incremental loading of TR with PG

- Every time a logic cone must be loaded into the solver, make a structural recursive visit of TR's AIG:
  - Carrying a flag that represents the polarity of the path:
    - Initialized with constrained value of output
    - Toggled every time an inverted edge is crossed
  - Load the right (left) side clauses of every gate that is reached by a {+} ({-}) path of recursion and that have not been loaded in that polarity yet
- **Percentage of TR that is needed per SAT query in average**:

SAT call type	% TR	% TR (PG)
Relative induction	52.8%	37.2%
Generalize	36.6%	26.5%
Propagation	40.6%	28.2%

  - About 30% reduction of logic cones
  - Using PG are solved **75 (68 +7)** instances of HWMCC 2012

Outline

- Preliminaries
- IC3 algorithm
- Characterization of SAT solving in IC3
- Incremental loading of transition relation
- **SAT solvers clean-up heuristics**
- Conclusions and future works

SAT solvers clean-up heuristics

- As verification proceeds clauses loaded from previous queries accumulate in solvers
  - Portions of previoulsy loaded TR's logic cones + deactivated clauses
  - The more clauses are loaded into the solver the slower BCP will be!
- Periodic **clean-ups** of the solvers are needed
  - IC3 performance degrades quickly without clean-ups
  - But they introduce some overhead:

    clauses must be reloaded into the solver + learning must be redone
- **Clean-up heuristics** try to find a tradeoff between clean-up overhead and BCP speedup

Clean-up heuristics

- Clean-up heuristics checks periodically if an **heuristic measure** *u* (estimate of the amount of "useless" clauses loaded in the solver) exceeds a given threshold *t*
  - if $u > t$ the solver is cleaned
- Two types of clean-up heuristics
  - **Static**: the threshold is a fixed value determined experimentally
  - **Dynamic**: the threshold varies dynamically in relation to some parameters of the solver

Cube-dependent utility

- Typically *u* corresponds to the number of deactivated clauses *a*
- **Cube-dependent utility**: based both on *a* and on the estimated size of useless loaded cones
  - $l(x_i')$: 0 if the logic cone of x_i' is not loaded, the number of clauses of that cone otherwise

$$u(cube) = a \quad + \sum_{x_i' \notin cube} l(x_i')$$

Evaluation of clean-up heuristics

- Three heuristics compared:
  - **(H1)** Static: $u = a > 300$
  - **(H2)** Dynamic: $u = a > \frac{1}{2}|variables|$ [Een, Mishchenko, Brayton 2011]
  - **(H3)** Dynamic: $u(cube) > \frac{1}{2}|clauses|$

SAT call type	H1 solving time	H2 solving time	H3 solving time
Relative induction	334 ms	1536 ms	707 ms
Generalize	575 ms	1877 ms	1039 ms
Propagation	681 ms	2426 ms	1397 ms

  - Surprisingly **H1** stands out as a clear winner!

Outline

- Preliminaries
- IC3 algorithm
- Characterization of SAT solving in IC3
- Incremental loading of transition relation
- SAT solvers clean-up heuristics
- **Conclusions and future works**

Conclusions and future work

- The use of the Plaisted-Greenbaum encoding in TR loading showed to be effective in reducing the size of loaded logic cones.
- **Some previously unsolved instances are now solved**
  - Can be profitably exploited in the context of **a portfolio-based approach**
- Our clean-up heuristic didn't proved to be effective, by now. Finding a tradeoff for solvers clean-up is not a trivial task.
  - Our research on the subject is still ongoing. It seems that cleaning-up solvers frequently achieves better results
- **Future works**:
  - Experiment different thresholds for the proposed clean-up heuristics
  - Investigate the use of specialized solver for expensive queries (generalization and propagation)
  - Collaborate with Berkley on the subject

Thank you!

Questions?