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

Gianpiero Cabodi (*), Alan Mishchenko (**), Marco Palena (*)

(*) Dip. di Automatica ed Informatica, Politecnico di Torino, Torino, Italy
(**) Dept. of EECS, University of California, Berkeley, CA, USA
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• SAT solvers clean-up heuristics
• Specialized solvers
• Experimental results
• Conclusions and future works
Outline

• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• SAT solvers clean-up heuristics
• Specialized solvers
• Experimental results
• Conclusions and future works
• **Boolean circuits modelled as** finite state transition systems:

\[ S = \langle x, I, T \rangle \]

- \( x = \{x_0, x_1, \ldots, x_n\} \): state variables.
- \( I(x) \): initial states.
- \( T(x, x') \): 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. \( x_1 \land \neg x_2 \land x_3 \)
  - **Clause** is a disjunction of literals: e.g. \( \neg x_1 \lor x_2 \lor x_3 \)
  - **CNF** is a conjunction of clauses: e.g. \((x_1 \lor \neg x_2 \lor x_3) \land (\neg x_2 \lor x_3)\)
• An assignment $s$ satisfies $F$ if $F$ evaluates to \textit{true} under $s$: $s \models F$

• $F$ is \textit{stronger} than $G$ if: $F \implies G$

• When an assignment $(s, t')$ satisfies $T$:
  – $s$ is a \textit{predecessor} of $t$
  – $t$ is a \textit{successor} of $s$

• $s_0, s_1, \ldots, s_n$ is a \textit{path} if $s_i, s_{i+1} \models T$ $\forall i : 0 \leq i < n$
• A state \( s \) is \textbf{reachable} if there exists a path \( s_0, s_1, \ldots, s_n = t : s_0 \models I \)
  
  - Set of \textit{n-bounded reachable states} of \( S \): \( R_n(S) \)
  
  - Set of \textit{reachable states} of \( S \):

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

• Given \( S = \langle x, I, T \rangle \) and property \( P(x) \) the \textbf{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: \( F \land T \rightarrow F' \)

• F is an **inductive invariant relative to G**:
  – Base case: \( I \rightarrow F \)
  – Inductive case: \( G \land F \land T \rightarrow F' \)

• An inductive invariant P is an over-approximation to reachable states \( \Rightarrow \) IVP can be seen as the problem to find an inductive invariant F stronger than the property P (inductive strengthening of P): \( F \rightarrow P \)
• Preliminaries

• **IC3 algorithm**

• Characterization of SAT solving in IC3

• Incremental loading of transition relation

• SAT solvers clean-up strategies

• Specialized solvers

• Experimental results

• Conclusions and future works
• Incremental SAT-based invariant verification algorithm that uses induction
• Maintains a set of over-approximations of bounded reachable states (time frames):
  \[ F_k = F_0, F_1, ..., F_k \]
  – Tries to find an inductive strengthening of P incrementally refining \( F_k \) with relative inductive clauses
• The following conditions hold throughout the algorithm:
  – \( F_0 = I \) \hspace{1cm} (C1)
  – For each \( 1 \leq i < k \):
    • \( F_i \rightarrow F_{i+1} \) \hspace{1cm} (C2)
    • \( F_i \land T \rightarrow F'_i \) \hspace{1cm} (C3)
    • \( F_i \rightarrow P \) \hspace{1cm} (C4)
IC3 algorithm

• At iteration $k$, IC3 enumerates states of $F_k$ that violate $P$:
  \[ \text{SAT?}[F_k \land \neg P] \]  
  \((Q1)\)

• 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 $I$)
• To block a cube \( c \) in \( F_k \), IC3 first tries to find out if \( \neg c \) is inductive relative to \( F_{k-1} \):

\[
\text{SAT?}[F_{k-1} \land \neg c \land T \land c']
\] (Q2)

• If not, a predecessor \( s \) is discovered \( \Rightarrow \) \( s \) must be blocked in \( F_{k-1} \) first \( \Rightarrow \) blocking of \( c \) is delayed, the procedure tries to block \( s \) in \( F_{k-1} \) \( \Rightarrow \) blocking procedure iterates
  – Eventually either \( \neg c \) become inductive relative to \( F_{k-1} \) or a predecessor in \( F_0 \) is found (path from initial states to a bad cube)
• If (Q2) is UNSAT, a clause $\neg c$ that is inductive relative to $F_{k-1}$ is found, then IC3 tries to remove literals from $\neg c$ to obtain an inductive generalization
  – Removing literals can break relative induction!
• For each literal removed, relative induction must be checked again:
  – **Inductive case**: $\text{SAT?}[F_{k-1} \land \text{cls} \land T \land \neg\text{cls}']$ (Q4)
  – **Base case**: $\text{SAT?}[I \land \neg\text{cls}]$ (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:
    $\text{SAT?}[F_k \land c]$ (Q3)
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 $F_{i+1}$

\[
\text{SAT}\left[ F_k \land T \land \neg\text{cls}' \right]
\]  \hspace{1cm} (Q6)

- If SAT, the clause cls is added to $F_{k+1}$

If during the propagation phase we discover that $F_i \equiv F_{i+1}$ for some $1 \leq i < k \implies F_i$ is an inductive strengthening for P
• Preliminaries
• IC3 algorithm

• **Characterization of SAT solving in IC3**
• Incremental loading of transition relation
• SAT solvers clean-up heuristics
• Specialized solvers
• Experimental results
• Conclusions and future works
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 (≈ $10^3$ – $10^6$)

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: \( SAT[F_i \land \neg P] \)
  
  (Q2) - Relative inductive check: \( SAT[F_i \land \neg \text{cube} \land T \land \text{cube}'] \)
  
  (Q3) - Blocked cube checks: \( SAT[F_i \land \text{cube}] \)
  
  (Q4) - Inductive generalization check: \( SAT[F_i \land \text{cls} \land T \land \neg \text{cls}'] \)
  
  (Q5) - Base of induction check: \( SAT[I \land \neg \text{cls}] \)
  
  (Q6) - Clause propagation check: \( SAT[F_i \land T \land \neg \text{cls}'] \)

- **HWMCC 2012** (PdT, time limit 900s, memory limit 2 GB): 79 solved instances/310

<table>
<thead>
<tr>
<th>SAT call type</th>
<th>% calls</th>
<th>Num calls</th>
<th>Solving time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target intersection</td>
<td>0.1%</td>
<td>483</td>
<td>81 ms</td>
</tr>
<tr>
<td>Relative induction</td>
<td>7.6%</td>
<td>31172</td>
<td>334 ms</td>
</tr>
<tr>
<td>Blocked cube</td>
<td>6.8%</td>
<td>27891</td>
<td>219 ms</td>
</tr>
<tr>
<td>Generalize</td>
<td>34.7%</td>
<td>142327</td>
<td>575 ms</td>
</tr>
<tr>
<td>Induction base</td>
<td>35.9%</td>
<td>147248</td>
<td>112 ms</td>
</tr>
<tr>
<td>Propagation</td>
<td>14.9%</td>
<td>61114</td>
<td>681 ms</td>
</tr>
</tbody>
</table>
• 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!**
    → Load as less clauses as needed
    → Clean-up periodically each solver
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
  • **Incremental loading of transition relation**
  • SAT solvers clean-up heuristics
  • Specialized solvers
• Experimental results
• Conclusions and future works
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
  \[\Rightarrow\] 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!
• Each SAT query that needs TR, constrains next states with a cube $c' \implies$ 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
• 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:**
  – About 20% reduction of logic cones
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• **SAT solvers clean-up heuristics**
• Specialized solvers
• Experimental results
• Conclusions and future works
• As verification proceeds clauses loaded from previous queries accumulate in solvers
  – Portions of previously 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
• Periodically check if a **heuristic measure** $u$ (estimate of the amount of “useless” clauses loaded into 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
• Typically $u$ corresponds to the number of deactivated clauses $a$
• **Cube-dependent heuristic measure**: based both on $a$ and on the estimated size of useless loaded cones
  – $A$: number of TR clauses already loaded into the solver (before loading the logic cones required by $c'$)
  – $S(c')$: size (number of clauses) of the logic cones required for solving the current query
  – $L(c')$: number of clauses in the required logic cones that must be loaded into the solver (equal to $S(c')$ minus the number of clauses that such cone shares with previously loaded cones)

$$u(c') = A - (S(c') - L(c'))$$
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• SAT solvers clean-up heuristics

• Specialized solvers
• Experimental results
• Conclusions and future works
Inductive generalization and clause propagation queries are by far the most expensive ones in terms of average SAT solving time.

To reduce the burden of each time frame’s SAT solver, we have experimented the use of specialized solvers for handling such queries.
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• SAT solvers clean-up heuristics
• Specialized solvers
  • Experimental results
• Conclusions and future works
• Tests conducted on the whole HWMCC’12 competition
• Time limit: 900 s, Memory limit: 2GB
• **PdTRAV:**
  – 63 \(\frac{(60 + 3)}{310}\) instances of HWMCC 2012 are solved
  – 16 instances less solved than our baseline version, but 3 previously unsolved problems are now solved
• **ABC:**
  – 79 \(\frac{(76 + 3)}{310}\) instances of HWMCC 2012 are solved
  – 1 instance less solved than our baseline version, but 3 previously unsolved problems are now solved
Tests conducted on 70 selected benchmarks from HWMCC’12
Time limit: 900 ms, Memory limit 2 GB
2 SAT solvers per time frame

<table>
<thead>
<tr>
<th>SAT call type</th>
<th># Solved</th>
<th># New</th>
<th>Avg Solving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>64</td>
<td>-</td>
<td>137 ms</td>
</tr>
<tr>
<td>Inductive Generalization Specialized</td>
<td>66</td>
<td>4</td>
<td>144 ms</td>
</tr>
<tr>
<td>Propagation Specialized</td>
<td>60</td>
<td>3</td>
<td>134 ms</td>
</tr>
</tbody>
</table>
Clean-up heuristics (PdTRAV)

- Four heuristics compared:
  - (H1) Static: $u = a > 300$
  - (H2) Dynamic: $u = a > \frac{1}{2}|variables|$
  - (H3) Dynamic: $avg\left(W\left(\frac{u(c')}{|clauses|}, 1000\right), 0.5\right) > 0.5$
  - (H4) Dynamic: $H2||H3$

- Tests on 70 selected benchmarks from HWMCC’12 (900s, 2 GB)

<table>
<thead>
<tr>
<th>SAT call type</th>
<th># Solved</th>
<th># New</th>
<th>Avg Solving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>64</td>
<td>-</td>
<td>137 ms</td>
</tr>
<tr>
<td>H2</td>
<td>60</td>
<td>1</td>
<td>122 ms</td>
</tr>
<tr>
<td>H3</td>
<td>44</td>
<td>0</td>
<td>116 ms</td>
</tr>
<tr>
<td>H4</td>
<td>62</td>
<td>3</td>
<td>125 ms</td>
</tr>
</tbody>
</table>
Different strategies were tested in conjunction with restart heuristic H4

Tests conducted on 70 selected benchmarks from HWMCC’12

Time limit: 900 ms, Memory limit 2 GB

<table>
<thead>
<tr>
<th>SAT call type</th>
<th># Solved</th>
<th># New</th>
<th>Avg Solving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>64</td>
<td>-</td>
<td>137 ms</td>
</tr>
<tr>
<td>H4 + PG</td>
<td>59</td>
<td>3</td>
<td>208 ms</td>
</tr>
<tr>
<td>H4 + Inductive Gen. Specialized</td>
<td>58</td>
<td>1</td>
<td>111 ms</td>
</tr>
<tr>
<td>H4 + PG + Inductive Gen. Specialized</td>
<td>50</td>
<td>1</td>
<td>178 ms</td>
</tr>
</tbody>
</table>
Tests on ABC

- Tests conducted on 70 selected benchmarks from HWMCC’12
- Time limit: 900 ms, Memory limit 2 GB
- ABC default restart strategy: \( u = a > 300 \)

<table>
<thead>
<tr>
<th>SAT call type</th>
<th># Solved</th>
<th># New</th>
<th>Avg Solving Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>64</td>
<td>-</td>
<td>138 ms</td>
</tr>
<tr>
<td>PG</td>
<td>63</td>
<td>1</td>
<td>152 ms</td>
</tr>
<tr>
<td>Without dynamic TR loading</td>
<td>63</td>
<td>2</td>
<td>158 ms</td>
</tr>
<tr>
<td>a &gt; 1000</td>
<td>64</td>
<td>1</td>
<td>138 ms</td>
</tr>
</tbody>
</table>
• Preliminaries
• IC3 algorithm
• Characterization of SAT solving in IC3
• Incremental loading of transition relation
• SAT solvers clean-up heuristics
• Specialized solvers
• Experimental results

• Conclusions and future works
Conclusions and future work

• Plaisted-Greenbaum
  – Less overall instances solved
  – Some previously unsolved instances are now solved

• Our attempts to define a more informed dynamic clean-up heuristic are still outperformed by a particularly tailored static heuristic
  – Our research on the subject is still ongoing. We believe that a more extensive experimentation could lead to a better parameter configuration

• Both approaches can be profitably exploited in the context of a portfolio-based approach

• Future works:
  – Experiment different parameters configurations for H4
Thank you!
Questions?