Model Checking with Boolean Satisfiability

Joao Marques-Silva
Electronics & Computer Science
University of Southampton

University of Swansea, CS Colloquia, December 2005
Goals of this talk

- SAT algorithms and concepts
- SAT-based model checking
- Utilization of interpolants in SAT-based model checking
- Optimizations to the utilization of interpolants
Outline

- Boolean Satisfiability (SAT)
- SAT-based model checking
- Improvements to SAT-based model checking
- Results
- Conclusions
Outline

- **Boolean Satisfiability (SAT)**
  - SAT algorithms
  - Resolution refutations
  - Interpolants

- SAT-based model checking

- Improvements to SAT-based model checking

- Results

- Conclusions
CNF formulas

- Conjunctive normal form (CNF):
  - Standard representation for SAT
  - CNF formula $\varphi$ is a conjunction of clauses
  - Clause is a disjunction of literals
  - Literal is a variable or its complement

$$\varphi = (a \lor b) \land (\neg a \lor c) \land (c \lor \neg d \lor \neg e) \land (\neg d \lor \neg a)$$

$$\varphi = (a \lor b) (\neg a \lor c) (c \lor \neg d \lor \neg e) (\neg d \lor \neg a)$$

- Can map propositional formulas into CNF in linear time
  - Addition of a linear number of auxiliary variables

[Tseitin’68; Plaisted&Greenbaum’86]
Given a partial assignment to the variables:

- A literal is satisfied if its value is 1; it is unsatisfied if its value is 0; otherwise it is unassigned.
- A clause is satisfied if at least one of its literals is satisfied; it is unsatisfied if all of its literals are unsatisfied; otherwise it is unresolved.
- A formula is unsatisfied if at least one clause is unsatisfied; it is satisfied if all clauses are satisfied; otherwise it is unresolved.

\[ \varphi = (a \lor b) \land (\neg a \lor c) \land (c \lor \neg d \lor \neg e) \land (\neg d \lor \neg a) \]
Algorithms for SAT

- **Incomplete Algorithms** *(Cannot prove unsatisfiability)*
  - Local search (hill climbing)
  - Lagrangian multipliers
  - Genetic algorithms
  - Simulated annealing
  - Tabu search
  - ...

- **Complete Algorithms** *(Can prove unsatisfiability)*
  - Backtrack search (DPLL)
  - Resolution
  - Stalmarck’s method
  - Recursive learning
  - Binary decision diagrams (BDDs)
  - ...

- The utilization of SAT in model checking requires ability to prove unsatisfiability
  - Most SAT algorithms used in model checking are based on backtrack search
Plain backtrack search

- Given a CNF formula $\varphi$, i.e. a conjunction of clauses, implicitly enumerate all partial assignments to the variables

Increasingly specified partial assignments

- No variables assigned
- All variables assigned

- Conflict: at least one unsatisfied clause
- Solution: all clauses satisfied
Unit propagation

- **Unit clause:**
  - A clause $\omega$ is unit iff all literals but one are assigned value 0 and one literal is unassigned
    - With $a = 0$ and $b = 1$, $\omega = (a \lor \neg b \lor c)$ is unit

- **Unit clause rule:**
  - If a clause $\omega$ is unit, then unassigned literal must be assigned value 1
    - With $a = 0$ and $b = 1$, $\omega = (a \lor \neg b \lor c)$ is unit
    - Literal $c$ must be assigned value 1 for $\omega$ to be satisfied
      - With $c = 1$, $\omega = (a \lor \neg b \lor c)$ becomes satisfied

- **Unit propagation:**
  - Iterative application of the unit clause rule
    - Imply variable assignments until no more unit clauses, or unsatisfied clause is identified
The DPLL algorithm

- **Backtrack search**
  - Implicit enumeration of all partial assignments

- **Unit propagation**
  - Iterated application of unit clause rule

- **Variable selection heuristic**
  - Policy for selecting the variable to branch on and the value to assign the variable

- **DPLL seldom used in practical applications until the mid 90s**!
Modern SAT algorithms

- Follow the organization of the DPLL algorithm [Davis et al.’62]
  - Backtrack search with unit propagation

- Several key techniques are used:
  - Clause learning [Marques-Silva&Sakallah’96]
    - Infer new clauses from causes of conflicts
      - Allows implementing non-chronological backtracking
  - Exploiting structure of conflicts [Marques-Silva&Sakallah’96]
    - Unique Implication Points (UIPs)
      - Dominators in graph of implied assignments
  - Optimised data structures [Moskewicz et al.’01]
    - Lazy evaluation of clause state
  - Adaptive branching heuristics [Moskewicz et al.’01]
    - Variable branching metrics are affected by number of conflicts
    - Aging mechanisms for focusing on most recent conflicts
  - Search restarts [Gomes,Selman&Kautz’98]
    - Opportunistically restart backtrack search
Clause learning

- During backtrack search, for each conflict learn clause that explains and prevents repetition of same conflict

\[ \phi = (a \lor b)(\neg b \lor c \lor d)(\neg b \lor e)(\neg d \lor \neg e \lor f) \ldots \]

Assume (decisions) \( c = 0 \) and \( f = 0 \)

Assign \( a = 0 \) and imply assignments

A conflict is reached: \((\neg d \lor \neg e \lor f)\) is unsatisfied

\((a = 0) \land (c = 0) \land (f = 0) \Rightarrow (\phi = 0)\)

\((\phi = 1) \Rightarrow (a = 1) \lor (c = 1) \lor (f = 1)\)

\[ \therefore \text{learn new clause: } (a \lor c \lor f) \]
Non-chronological backtracking

- During backtrack search, in the presence of conflicts, backtrack to one of the causes of the conflict.

\[
\varphi = (a \lor b)(\neg b \lor c \lor d)(\neg b \lor e)(\neg d \lor \neg e \lor f)
\]

\[
(a \lor c \lor f)(\neg a \lor g)(\neg g \lor b)(\neg h \lor j)(\neg i \lor k)\ldots
\]

Assume (decisions) \( c = 0, f = 0, h = 0 \) and \( i = 0 \)

Assignment \( a = 0 \) caused conflict \( \Rightarrow \) learned clause \( (a \lor c \lor f) \)

\( (a \lor c \lor f) \) implies \( a = 1 \)

A conflict is again reached: \( (\neg d \lor \neg e \lor f) \) is unsatisfied

\( (c = 0) \land (f = 0) \Rightarrow (\varphi = 0) \)

\( (\varphi = 1) \Rightarrow (c = 1) \lor (f = 1) \)

\[.:. \text{learn new clause: } (c \lor f)\]
Non-chronological backtracking

Learned clause \((c \lor f)\)

Need to backtrack, given \((c \lor f)\)

Backtrack to most recent decision: \(f = 0\)

\[\therefore \text{ Clauses learned: } (a \lor c \lor f) \text{ and } (c \lor f)\]

In practice, learned clauses can allow backtracking over a significant percentage of the decision variables
## Evolution of SAT solvers

- Remarkable improvements over the last decade

<table>
<thead>
<tr>
<th>Instance</th>
<th>Posit' 94</th>
<th>Grasp' 96</th>
<th>Chaff'01</th>
<th>Siege'04</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa2670-136</td>
<td>28.53</td>
<td>0.36</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>bf1355-638</td>
<td>772.45</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>design_1</td>
<td>&gt;7200</td>
<td>65.35</td>
<td>1.27</td>
<td>0.29</td>
</tr>
<tr>
<td>design_3</td>
<td>&gt;7200</td>
<td>9.13</td>
<td>0.52</td>
<td>0.41</td>
</tr>
<tr>
<td>f_ind</td>
<td>&gt;7200</td>
<td>4663.89</td>
<td>17.91</td>
<td>6.52</td>
</tr>
<tr>
<td>splitter_42</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
<td>28.81</td>
<td>4.46</td>
</tr>
<tr>
<td>c6288</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
<td>2847.46</td>
</tr>
<tr>
<td>pipe_64_32</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
<td>&gt;7200</td>
</tr>
</tbody>
</table>
Resolution

- Refutation-complete procedure for first order logic

- In propositional logic:
  - Technique for deriving new clauses
    - Example: $\omega_1 = (\neg a \lor b \lor c)$, $\omega_2 = (a \lor b \lor d)$
    - Resolution:
      \[
      \text{res}(\omega_1, \omega_2, a) = (b \lor c \lor d)
      \]
  - Forms the basis of a complete procedure for satisfiability
  - Impractical for real-world formulas
  - Application of restricted forms has been successful
    - E.g., restricted resolution
      \[
      \text{res}((\neg a \lor \alpha), (a \lor \alpha), a) = (\alpha)
      \]
      \[\alpha\] is a disjunction of literals

[Robinson’65]
[Davis&Putnam’60]
Resolution refutations

- Clause learning can be viewed as the inference of a clause by a sequence of resolution steps

\[ \phi = (a \lor b)(\neg b \lor c \lor d)(\neg b \lor e)(\neg d \lor \neg e \lor f) \ldots \]

- \( a = 0 \) yields conflict; can learn \((a \lor c \lor f)\)
- By applying resolution:

\[ \phi = (a \lor b)(\neg b \lor c \lor d)(\neg b \lor e)(\neg d \lor \neg e \lor f) \ldots \]
Deriving resolution refutations

- For unsatisfiable formulas:
  - Learned clauses capture a resolution refutation from a subset of the original clauses
  - SAT solvers can be instructed to recreate resolution refutation for unsatisfiable formula

\[ \varphi = (a \lor b) (\neg a \lor c) (\neg b) (\neg c) \]

\[ \omega_1 \quad \omega_2 \quad \omega_3 \quad \omega_4 \]

\[ a = 0 \quad c = 0 (\omega_4) \]

\[ b = 0 (\omega_3) \]

\[ \kappa \]

\[ (a \lor b) \quad (\neg a \lor c) \]

\[ (b \lor c) \quad (\neg b) \]

\[ (c) \quad (\neg c) \]

\[ \bot \]

[Zhang&Malik'03]
Interpolants

Given two subsets of clauses $A$ and $B$, assume $A \land B$ is unsatisfiable. Then, there exists an interpolant $A'$ for the pair $(A, B)$ with the following properties:

- $A$ implies $A'$
- $A' \land B$ is unsatisfiable
- $A'$ refers only to the common variables of $A$ and $B$
- Example:
  - $A = p \land q$, $B = \neg q \land r$
  - $A' = q$

Recent result:

- Given a resolution refutation of $A \land B$, can compute interpolant for the pair $(A, B)$ in linear time on the size of the resolution refutation
  - SAT solvers can be instructed to output resolution refutation!

Computing interpolants:

- Different algorithms can be used
  - Pudlak’97, McMillan’03

[Craig’57]

[Pudlak’97]

[McMillan’03]
Computing interpolants

\[ A = (r \lor y)(\neg r \lor x) \]
\[ B = (\neg y \lor a)(\neg y \lor \neg a)(\neg x) \]

- Interpolant is a Boolean circuit that follows structure of resolution refutation
  - Can map circuit into CNF in linear time and space

\[ A' = y + x \]

A implies A'; A' \land B is unsatisfiable
A' with variables common to A and B

[Tseitin'68; Plaisted&Greenbaum'86]
Outline

- Boolean Satisfiability (SAT)
  - SAT-based model checking
    - Bounded model checking
    - Unbounded model checking
      - Utilization of interpolants
  - Improvements to SAT-based model checking
- Results
- Conclusions
Verification by model checking

• *Given*,
  – A model $M$ of a system
  – A property $\phi$
    Prove that $M$ satisfies $\phi$: $M \models \phi$

• Property specified using some **temporal logic**
  – Computation tree logic (CTL)
    • Discrete time, branching time
    • Often used for hardware systems
  – Other temporal logics: LTL, CTL*

• Types of properties:
  – **Safety**: Some unwanted condition will not occur
  – **Liveness**: Some expected condition will eventually take place
Representing systems

- Fundamental entities are **states**
- Dynamic behaviour captured in terms of state transitions
  - Transition relation:
    - Binary relation on the set of states $S$: $\rightarrow \subseteq S \times S$
    - $s \rightarrow s'$ provided the system can reach $s'$ from $s$ in one computation step
      - Can be represented with a characteristic function $T(s,s')$
- Atomic propositions can label each state $s$, $L(s)$
  - Denoting which atomic propositions are true at each state $s$

- Transition system (or model) $M = (S, \rightarrow, L)$:
  - Set of states: $S$
    - May consider a set of initial states $S_0$
  - Binary relation describing state transitions: $\rightarrow$
  - Labelling function: $L : S \rightarrow P(\text{Atoms})$
    - $L(s)$: atomic propositions that are true in state $s$
    - Maps $S$ into the power set of the atoms
An example

\[ S = \{ s_0, s_1, s_2 \} \]

\[ \rightarrow = \{ (s_0,s_1), (s_0,s_2), (s_1, s_2), (s_1, s_0), (s_2, s_2) \} \]

\[ L(s_0) = \{ p, q \} \]
\[ L(s_1) = \{ q, r \} \]
\[ L(s_2) = \{ r \} \]
Computation Tree Logic (CTL) [Clarke&Emerson'81]

- Propositional logic with temporal operators:
  - \( \phi := \bot \mid p \mid (\neg \phi) \mid (\phi \land \phi) \mid (\phi \lor \phi) \mid (\phi \rightarrow \phi) \mid A X \phi \mid E X \phi \mid A G \phi \mid E G \phi \mid A F \phi \mid E F \phi \mid A [\phi U \phi] \mid E [\phi U \phi] \)
  - Standard logical connectives
  - Temporal operators:
    - A: along all computation paths
    - E: along at least one path (exists)
    - X: next state
    - F: some future state
    - G: all future states (globally)
    - U: until
  - Adequate set of operators: \(\neg, \land, E X, A F, E U\)

- Types of properties:
  - Safety: \(A G \phi\)
  - Liveness: \(A F \phi\)
  - ...

- Can formalise semantics of CTL
CTL model checking algorithms

- **Iterative state marking**  
  - Impractical for transition systems with a large number of states

- **BDD-based symbolic procedure**  
  - Implicit manipulation of sets of states
  - Allowed the widespread use of CTL model checking
  - Number of state variables needs to be “small” enough

- **Utilization of SAT**  
  - Scales better than BDDs
  - Difficult to manipulate sets of states

References:
- Queille & Sifakis (1981)
- Burch et al. (1990)
- Biere et al. (1999), Sheeran et al. (2000)
SAT-based model checking

- Exploit robustness of modern SAT algorithms in model checking
  - Bounded model checking (BMC)
    - Focus on safety properties \( A G p \)
      - Where \( p \) is propositional
    - Useful for finding counterexamples
      - Hence proving \( E F \neg p \)
    - Usually incomplete
  - Unbounded model checking (UMC)
    - Also focus on safety properties \( A G p \)
    - Objective is to ensure completeness
      - Either provide counterexample, or prove property
Bounded model checking

- Safety properties: \( \mathcal{E} \mathcal{F} \neg p \)

\[
\Phi^k = I_0(s_0) \land \bigwedge_{j=0}^{k-1} T(s_j, s_{j+1}) \land \left( \bigvee_{j=r}^{k} \neg p_j \right)
\]

- Characteristic functions for representing initial states and transition relation, respectively \( I_0 \) and \( T \)
  - Resulting CNF formula: \( I_0 \land U_k \land F_k \)
    - Where:
      \[
      U_k = \bigwedge_{j=0}^{k-1} C_j, \quad F_k = \left( \bigvee_{j=r}^{k} \neg p_j \right)
      \]
  - Interpretation:

[Biere et al.'99]
An example

- Property: $M, s_0 \models A G \neg q$?
- Evaluate: $M, s_0 \models E F q$
- Unroll model $k$ time steps:

- Check satisfiability of CNF formula for $I_0 \land U_k \land F_k$
Bounded model checking

- A possible BMC algorithm:
  - Given some initial k
  - While $k \leq$ user-specified time-bound $B$
    - Generate CNF formula for $I_0 \land U_k \land F_k$
    - Invoke SAT solver
    - If formula is satisfiable, then a counterexample within $k$ time steps has been found
      - Return counterexample
    - Otherwise, increase $k$

- The BMC algorithm is **incomplete**
  - But it is complete if recurrence diameter is known!
Towards completeness

- Unbounded model checking
  - Utilization of induction
    - Standard BMC loop
      - Stop BMC loop for some $i$, if cannot have loop-free path of size $i$ that can be reached from $I_0$
      - All distinct states that are reachable from $I_0$ have been accounted for
    - Maximum unfolding bounded by largest loop-free path
  - ... [Sheeran et al.'00]

- Utilization of interpolants [McMillan'03]
  - BMC and Craig interpolants allow SAT-based computation of abstractions of reachable states
    - Avoid computing exact sets of reachable states
    - The most promising approach in practice
      - Maximum unfolding bounded by largest shortest path between any two states
Abstraction of reachable states

- For each iteration of BMC loop, call to SAT solver returns unsat until counterexample is found
  - Analysis of resolution refutation can yield abstractions of reachable states
    \[ \Phi = I_0 \land C_0 \land C_1 \land \ldots \land C_{k-1} \land F_k = A \land B \]
    \[ A = I_0 \land C_0 \]
    \[ B = C_1 \land \ldots \land C_{k-1} \land F_k \]

- Given A and B, and a resolution refutation for \( A \land B \), compute Craig interpolant \( A' \):
  - \( A = I_0 \land C_0 \) implies \( A' \)
  - \( A' \land B \) is unsatisfiable
  - \( A' \) solely represented with state variables
    - If \( A \) holds, then \( A' \) holds
      - \( A_1 = A' \) represents abstraction of states reachable from \( I_0 \) in 1 time step!
Fixpoint of reachable states

- Can iterate computation of interpolants:

If $A_i \rightarrow I_0 \lor A_1 \lor A_2 \lor \ldots \lor A_{i-1}$, then a fixpoint is reached; all reachable states identified!
If \( F_k \) is satisfied from \( I_0 \), then we have a counterexample!

If a fixpoint of the reachable states is identified, then no reachable state can satisfy property! If \( A \land B \) is sat, may have abstracted too much; must unfold more time steps.

Maximum value of \( k \) is bounded by largest shortest path between any two states.
Outline

- Boolean Satisfiability (SAT)
- SAT-based model checking
  - Improvements to SAT-based model checking
    - Representation of interpolants
    - Rescheduling BMC & UMC loops
- Results
- Conclusions
Redundancy in interpolants

\[ A = (r \lor y)(\neg r \lor x) \]

\[ B = (\neg y \lor a)(\neg y \lor \neg a)(\neg x) \]

\[ (r \lor y) \quad (\neg r \lor x) \]

\[ (y \lor x) \quad (\neg y \lor a) \quad (\neg y \lor \neg a) \]

\[ (\neg x) \quad (\neg y) \quad (x) \]

\[ A' = y + x \]

Must simplify circuit to get \( A' = y + x \)
Redundancy in interpolants

- Use dedicated representation:
  - Reduced Boolean Circuits (RBCs)
    - Compact representation
    - Polynomial
      - Non-canonical
    - Quite effective in practice
  - Binary Expression Diagrams (BEDs)
  - Binary Decision Diagrams (BDDs)

[Abdulla et al.'00]
Rescheduling the UMC loop

let $k = 0$
repeat
  if from $I_0$ can satisfy $F_k$ within $k$ steps
    return reachable
  $R = I_0$
  let $A = I_0 \land C_0$, and $B = C_1 \land C_2 \land \ldots \land C_{k-1} \land F_k$
  while $A \land B = \text{false}$
    $P = \text{unsat\_proof}(A \land B)$
    $A' = \text{interpolant}(P, A, B)$
    if $A' \rightarrow R$, return unreachable
    $R = A' \lor R$
    $A = A' \land C_0$
  end while
  increase $k$
end repeat

BMC loop
Number of iterations can be used to restrict when to call the next fixpoint check!
Rescheduling the UMC loop

Fixpoint checking with \( i \) iterations:

\[
\text{while } A \land B = \text{false} \\
P = \text{unsat\_proof}(A \land B) \\
A' = \text{interpolant}(P, A, B) \\
\text{if } A' \rightarrow R, \text{ return unreachable} \\
R = A' \lor R \\
A = A' \land C_0 \\
\text{end while}
\]

\begin{align*}
I_0 & \rightarrow A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_i \\
\end{align*}

Checked all states reachable in up to \( k+i-1 \) states, with an unfolding of size \( k \)

\[
\therefore \text{Need to check } \textbf{fixpoint} \text{ condition } \textbf{only} \text{ when unfolding of FSM exceeds } k+i-1 \text{ time steps}
\]
Rescheduling the BMC loop

let k = 0
repeat
  if from $I_0$ can satisfy $F_k$ within k steps
    return reachable
  R = $I_0$
  let $A = I_0 \land C_0$, and $B = C_1 \land C_2 \land \ldots \land C_{k-1} \land F_k$
  while $A \land B = \text{false}$
    $P = \text{unsat\_proof}(A \land B)$
    $A' = \text{interpolant}(P, A, B)$
    if $A' \rightarrow R$, return unreachable
    $R = A' \lor R$
    $A = A' \land C_0$
  end while
  increase k
end repeat

BMC loop

Number of iterations can also be used to restrict when to check the BMC condition!

Fixpoint
Rescheduling the BMC loop

while $A \land B = false$

$P = \text{unsat\_proof}(A \land B)$

$A' = \text{interpolant}(P, A, B)$

if $A' \rightarrow R$, return unreachable

$R = A' \lor R$

$A = A' \land C_0$

end while

Fixpoint checking with $i$ iterations:

$I_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_i$

Checked all states reachable in up to $k+i-1$ states, with an unfolding of size $k$

$\therefore$ Need to check BMC condition only when unfolding of FSM exceeds $k+i-1$ time steps
Outline

- Boolean Satisfiability (SAT)
- SAT-based model checking
- Improvements to SAT-based model checking

- Results
- Conclusions
Our model checker MCSAT

- SAT-based model checker for safety properties

- Bounded model checking
  - Tight integration with SAT solver (CQuest, ...)
    - Incremental generation of CNF formula
    - Reutilization and replication of learned clauses

- Unbounded model checking
  - Utilization of interpolants
    - Additional improvements
      - Fast algorithms for computing interpolants
      - Rescheduling fixpoint checking and BMC loop given feedback from checking existence of a fixpoint
Experimental results

- Evaluated optimizations on set of examples
  - Specifically designed and industrial examples
- Evaluated both the plain UMC algorithm and the proposed optimizations

<table>
<thead>
<tr>
<th>Instance</th>
<th>No-reschedule</th>
<th>Reschedule BMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit counter</td>
<td>0.31</td>
<td>0.09</td>
</tr>
<tr>
<td>5-bit counter</td>
<td>3.86</td>
<td>0.84</td>
</tr>
<tr>
<td>6-bit counter</td>
<td>21.36</td>
<td>10.41</td>
</tr>
<tr>
<td>7-bit counter</td>
<td>1780.68</td>
<td>175.69</td>
</tr>
<tr>
<td>I12</td>
<td>255.77</td>
<td>272.47</td>
</tr>
<tr>
<td>I11</td>
<td>75.28</td>
<td>81.89</td>
</tr>
<tr>
<td>I31</td>
<td>83.51</td>
<td>90.08</td>
</tr>
<tr>
<td>I32</td>
<td>19.66</td>
<td>14.89</td>
</tr>
<tr>
<td>I33</td>
<td>17.44</td>
<td>13.09</td>
</tr>
<tr>
<td>I21</td>
<td>24.93</td>
<td>26.48</td>
</tr>
<tr>
<td>Total time</td>
<td>2282.8</td>
<td>685.9</td>
</tr>
</tbody>
</table>
Conclusions

- SAT technology has improved dramatically over the last decade
  - Key techniques:
    - Clause learning, optimised data structures, adaptive branching heuristics

- SAT has been applied to CTL model checking with success
  - Bounded and unbounded model checking

- Described optimizations to the utilization of interpolants in SAT-based model checking

- Results promising, but for specific instances
  - Rescheduling can allow number of iterations to be significantly reduced
Deriving resolution refutations

- For unsatisfiable formulas:
  - Learned clauses capture a resolution refutation from a subset of the original clauses
  - SAT solvers can be instructed to recreate resolution refutation for unsatisfiable formula

\[ \varphi = (a \lor b) (\neg a \lor c) (\neg b) (\neg c) \]

\( \omega_1 \quad \omega_2 \quad \omega_3 \quad \omega_4 \)

\[ a = 0 \quad b = 0 \quad c = 0 \]

\( \kappa \quad \omega_1 \quad \omega_2 \quad \omega_1 \quad \omega_1 \quad \omega_3 \quad \omega_4 \)

\[ \neg c \quad \neg b \quad c \quad (b \lor c) \quad (a \lor b) \quad (\neg a \lor c) \]

\[ \bot \]

[Zhang&Malik’03]
Bounded model checking

- **Given a bound** \( k \)
  - Number of computation steps, clock cycles, etc.

- **And a representation of the initial states**
  - \( I_0 \) is the characteristic function for the set of initial states
    - \( I_0(s) = 1 \) iff \( s \) is an initial state

- **Unroll the transition relation for** \( k \) **computation steps:**
  \[
  U_k = \bigwedge_{i=0}^{k-1} C_i
  \]

Where,
- \( C_i \) is the characteristic function for the \( i^{th} \) replica of the transition relation: \( s \rightarrow s' \)
  - \( C_i(s, s') = 1 \) iff \( s \rightarrow s' \)

[Biere et al.'99]
Bounded model checking

- Unroll the (safety) temporal property $k$ computation steps, $F_k$
  - Property describes a condition that should not hold in any of the $k$ computation steps

- Form the conjunction of the initial states condition, the unrolled transition relation, and the property:
  - $I_0 \land U_k \land F_k$

- Create instance of SAT for $I_0 \land U_k \land F_k$
  - E.g. apply Tseitin’s transformation to create a CNF formula

- Solve with SAT solver
Computing interpolants

- **Plain solution**
  - Trace learned clauses
    - Keep dependencies of each learned clause
  - Construct resolution proof
  - Generate interpolant

- **Optimised solution**
  - Generate interpolant directly from the trace, and skip the generation of the resolution proof
  - Interpolants are highly redundant Boolean expressions and can be extensively simplified
    - Simplify sub-expressions as early as possible
    - Visit clause dependencies depth-first (instead of breadth-first)
      - More suitable for early simplification of sub-expressions of the interpolant
      - Problem: must keep proof trace in memory