Boolean Satisfiability Solvers: Techniques and Extensions

Sharad Malik
Georg Weissenbacher
Princeton University

MOD 2011 Summer School, Bayrischzell
Exercises/Solutions on MOD 2011 Website

Exercises refer to the notes.
SAT in a Nutshell

- Given a propositional logic (Boolean) formula, find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.

\[ F = (a + b)(a' + b' + c) \]

- For \( n \) variables, there are \( 2^n \) possible truth assignments to be checked.

- First established NP-Complete problem.

SAT and more Expressive Formalisms

- Propositional Logic is a subset of
  - First Order Logic
  - Higher-Order Logic

- Validity of propositional formulas is decidable
  - First Order Logic: semi-decidable
    (by Gödel’s completeness theorem)
  - Arithmetic and Higher-Order Logic: undecidable
    (by Gödel’s incompleteness theorem)

- Complexity of SAT: NP-complete
  - But often tractable in practice
Where are we today?

- Intractability of the problem no longer daunting
  - Can regularly handle practical instances with millions of variables and constraints
- SAT has matured from theoretical interest to practical impact
  - Electronic Design Automation (EDA)
    - Widely used in many aspects of chip design
  - Increasing use in software verification
    - Commercial use at Microsoft, NEC,…
  - CAV 2009 Award for industrial impact
Welcome to SAT Live!

If you are a newcomer to the SATisfiability problem, you might want to take a look at Wikipedia’s page on the boolean satisfiability problem first. You might also find those surveys insight of the current interest on SAT solvers for software and hardware verification. Armin's there course on formal systems is a good start. Eugene Goldberg has also a nice and of introducing modern SAT solvers in his three part course on SAT. Finally, Ivo Marques Silva wrote a nice article on practical applications of boolean satisfiability.

Looking for a SAT solver to play with? The following open source SAT solvers might be a good start: Minisat (C++), Picosat (C), SAT4J (Java). If you are looking for a stochastic local search framework, take a look at UbSAT.

You can take a look at all the current links, see the links classified by keywords or add your own reference (you must be subscribed to SAT Live! or propose it as anonymous).

If you don’t have some links to propose for now but would like email notification of new additions to the repository, you can subscribe to the SAT Live! notification list or register to the site RostiMuss, using Rapper.

Finally, a page with some people interested by the SATisfaction problem is also available.

Last 10 new entries

<table>
<thead>
<tr>
<th>Date</th>
<th>Title</th>
<th>Hits</th>
<th>Contributed by</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-Jun-2011</td>
<td>Offer for a PhD position or a Post-doc position</td>
<td>18</td>
<td>Stephan Eppersorg</td>
<td>Job</td>
</tr>
<tr>
<td></td>
<td>Innovative approaches to guarantee correctness while designing embedded systems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group of Computer Architecture headed by Prof. Dr. Rolf Drechsler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>University of Bremen, Bremen, Germany</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Application</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The deadline for applications is July 10th 2011. Applications including CV, certificates, and recommendation letters should be sent by email to Rolf Drechsler (<a href="mailto:drechsler@uni-bremen.de">drechsler@uni-bremen.de</a>). The employer number is 23/11.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Salary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dependent on the qualification of the applicant the salary grade for the position as a researcher (Wissenschaftliche/r MitarbeiterIn) will be TVL 13 or TVL 14, i.e. net income 1800 EUR or 2000 EUR, respectively. The project will start on August 1st 2011.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The internationally renowned Group of Computer Architecture at the University of Bremen develops design automation tools for circuits and systems. Focus of the offered development of innovative approaches to guarantee correctness while designing embedded systems. The position is part of a research project funded by the German Research Foundation (DFG) for 5 years within a Reinhart Koselleck-Project.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The research group tightly cooperates with industrial partners within transfer projects, funded e.g. by the German Ministry for Education and Research (BMBF). Within the</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SAT Competition 2011

A competitive event of the SAT 2011 Conference
June 19th - June 22nd 2011, Ann Arbor, MI, USA

Last modification: Sat, 2011-04-23 15:14:21 GMT

Quick links
- Registration
- What's new this year?
- Competition tracks
- Submissions
- Important dates
- Judges
- Organizers
- Sponsors

Registration
Register and submit your solver or benchmark

What's new this year?

There are several new features in the SAT competition this year:

New Hardware
The competition will run on a new cluster at CRIL, composed of nodes with two Bi-Xeon Quad core processors and 32 GB of RAM. The operating system is CentOS 5.4, x86.

Sequential/Parallel Neutrality
- This year, there is no special track dedicated to sequential or parallel solvers. Sequential and parallel solvers are grouped into one single competition, but with two different ranking criteria: wall clock time and CPU time.
- This criterion allows for a more diverse range of SAT solvers to be compared. The best solvers will be awarded only one sequential solver will be launched on each processor (2 solvers per node) launched on a node (1 solver per node), with a memory limit of 15GB of RAM.

Sequential/Parallel Neutrality
- This year, there is no special track dedicated to sequential or parallel solvers. Sequential and parallel solvers are grouped into one single competition, but with two different ranking criteria: wall clock time and CPU time.
- This criterion allows for a more diverse range of SAT solvers to be compared. The best solvers will be awarded only one sequential solver will be launched on each processor (2 solvers per node) launched on a node (1 solver per node), with a memory limit of 15GB of RAM.

New Award Categories
- The competition will award both the fastest SAT solvers in terms of wall-clock time and in terms of CPU time. The most innovative ("non CDCL") SAT solver will be awarded a special award.
- Choose Your Category
- Unlike the previous competitions, in which all solvers were ranked by CPU time alone, the competition will now use wall clock time and CPU time to rank solvers.

Minimally Unsatisfiable Subset (MUS) Special Track
- Due to the success of MUS techniques on various applications (especially as core engines in MAXSAT solvers), a special track for MUS systems will be organized for the first time.

Data Analysis Track
- Since there are many different ways to analyze the results of the competition, the Data Analysis Track will offer anyone the possibility to run its own analysis of the competition the competition website and as a poster during the SAT conference. This track is an opportunity to experiment different ranking schemes, as well as analyze the strengths and weaknesses of the solvers.

Competition tracks

Here is a quick view of the competition. See detailed rules for complete details.

Main track
Where are we today? (contd.)

- Significant SAT community
  - SatLive Portal and SAT competitions
  - SAT Conference
- Emboldened researchers to take on even harder problems related to SAT
  - Max-SAT
  - Satisfiability Modulo Theories (SMT)
  - Quantified Boolean Formulas (QBF)
SAT Solvers in the Verification Context

- (static) program verification
- interpolation
- proof theory
- resolution
- complexity
- temporal logic
- equivalence checking
- BMC
- symbolic simulation
- model checking
- decision procedures
- SMT
- predicate abstraction
- compiler correctness
- complexity
Applications: Decision Procedures

- Bit-Vector Arithmetic
- “Bit-Flattening” of bit-vector operations

(a) Ripple carry adder
(b) Full adder
Applications: Hardware Verification

- Hardware verification
- Enabling technique for Bounded Model Checking

Can be encoded as Boolean Formula

Huffman Model
Applications: Bounded Model Checking

- Analyze fixed number of execution steps/cycles
  - Verification of temporal logic properties
    (e.g., “eventually every request is acknowledged”)
Applications: Software Verification

- Bounded Model Checking also used for Software
- Unwinding of program (control flow graph)
Applications: Equivalence Checking

- Check whether two programs are equivalent
  - Equivalence of behavior of SystemC/Verilog designs
  - Correctness of code optimizations

\[
x = 2y + z \quad \quad \quad \quad \quad t = y \ll 1;
\]
\[
x = t + z;
\]
SAT: Techniques and Extensions

- Part I: SAT Solving Techniques
- Part II: Extensions
- Part III: Example Application
SAT Solvers: A Condensed History

- **Deductive**
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”

- **Backtrack Search**
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment

- **Conflict Driven Clause Learning [CDCL]**
  - GRASP: Integrate a constraint learning procedure, 1996

- **Locality Based Search**
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation

- **“Pre-processing”**
  - Peephole optimization, e.g. miniSAT, 2005
Problem Representation

- **Conjunctive Normal Form**
  - Representation of choice for modern SAT solvers

\[(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')\]
Circuit to CNF Conversion

- **Tseitin Transformation**

\[
\begin{align*}
\text{d} & \equiv (a + b) \\
(a + b + d') \\
(a' + d) \\
(b' + d)
\end{align*}
\]

\[
\begin{align*}
\text{e} & \equiv (c \cdot d) \\
(c' + d' + e) \\
(d + e') \\
(c + e')
\end{align*}
\]

Consistency conditions for circuit variables

- **Can ‘e’ ever become true?**

Is \((e)(a + b + d')(a'+d)(b'+d)(c'+d+e)(d+e')(c+e')\) satisfiable?
Resolution

- Resolution of a pair of distance-one clauses

\[(a + b + c' + f) \land (g + h' + c + f)\]

Resolvent implied by the original clauses

\[a + b + g + h' + f\]
Resolution

- Resolution corresponds to quantifier elimination

\[
\exists x \cdot (C' + x) \cdot (D + \overline{x}) \\
\equiv ((C + x) \cdot (D + \overline{x})) [x/1] + ((C + x) \cdot (D + \overline{x})) [x/0] \\
\equiv (C + 1) \cdot (D + \overline{1}) + (C + 0) \cdot (D + \overline{0}) \\
\equiv C + D
\]

- Resolution calculus is refutation complete
  - Can always derive (\) if CNF formula is unsatisfiable
Davis Putnam Algorithm


- Iterative existential quantification of variables

Potential memory explosion problem!
Davis-Putnam Algorithm

- **1-literal-rule:**
  If formula contains unit clause \((e)\)
  - Set \(e\) to 1
  - Remove literal \(e'\) from all other clauses
  - Remove all clauses containing \(e\)

- **Affirmative-negative-rule:**
  If literal \(e\) occurs only positively or only negatively
  - Remove all clauses containing \(e\) (or \(e'\), respectively)

- For the remaining clauses use resolution
  - \((C+e)(D+e')=(C+D)\)
SAT Solvers: A Condensed History

- **Deductive**
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”

- **Backtrack Search**
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment

- **Conflict Driven Clause Learning [CDCL]**
  - GRASP: Integrate a constraint learning procedure, 1996

- **Locality Based Search**
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation

- **“Pre-processing”**
  - Peephole optimization, e.g. miniSAT, 2005
Basic DLL Search

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c)
\end{align*}
\]

Basic DLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DLL Search

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (a' + b + c) \]
\[ (a' + b' + c) \]
Basic DLL Search

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
b' + c' + d \\
(a' + b + c') \\
(a' + b' + c)
\end{align*}
\]
Basic DLL Search

\[ \begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c)
\end{align*} \]

\[ \begin{align*}
\rightarrow (a + c' + d) \\
\rightarrow (a + c' + d') \\
\rightarrow (b' + c' + d) \\
\rightarrow (a' + b + c') \\
\rightarrow (a' + b' + c)
\end{align*} \]
Basic DLL Search

Implication Graph

Unit Clause Rule

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Unit

(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Unit Clause Rule

a=0

d=1

c=0

d=1

Implication Graph
Basic DLL Search

\[(a' + b + c)\]
\[\rightarrow (a + c + d)\]
\[\rightarrow (a + c + d')\]
\[\leftarrow Unit\]
\[d = 1, d = 0\]

Implication Graph
Basic DLL Search

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (a' + b + c) \]
\[ (b' + c' + d) \]
\[ (a' + b + c') \]
\[ (a' + b' + c) \]

Conflict! Implication Graph

\[ d=1, d=0 \]

Implication Graph

Conflict!
Basic DLL Search

- $(a' + b + c)$
- $(a + c + d)$
- $(a + c + d')$
- $(a + c' + d)$
- $(a + c' + d')$
- $(b' + c' + d)$
- $(a' + b + c')$
- $(a' + b' + c)$

Diagram:
- Node a
- Node b
- Node c
- Arrows indicate search direction
- Backtrack arrow from c to a

Note: The image includes a visual representation of the DLL search process, with nodes and arrows indicating the search trajectory.
Basic DLL Search

(a' + b + c)
(a' + c + d)
(a + c + d')
(a' + c' + d)
(a' + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Forced Decision

Diagram:
- Node a
  - 0
- Node b
  - 0
- Node c
  - 0
  - 1
- Red box
  - ← Forced Decision
Basic DLL Search

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a' + b + c)\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

Conflicts!

Implication Graph

Forced Decision

\[d=1, d=0\]
Basic DLL Search

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (a' + b + c) \]
\[ (a' + b' + c) \]

Diagram:

- **a**
  - **b**
  - **c**
    - **0**
    - **1**
  - **Backtrack**

- (a' + b + c')
- (a' + b' + c)
Basic DLL Search

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Backtrack

Diagram:

- Node a
  - Edge 0 to node b
- Node b
  - Edge 0 to node c
  - Edge 1 to node c
- Node c
  - Edge 0 to node b
- Node b
  - Edge 0 to node a
Basic DLL Search

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DLL Search

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(a' + b + c)\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

\[d=1\]
\[c=0\]
\[a=0\]
\[c=0\]
\[d=1\]
\[d=0\]

Decision

Implication Graph

Conflict!
Basic DLL Search

\[
(a' + b + c)
\]

\[
\rightarrow (a + c + d)
\]

\[
\rightarrow (a + c + d')
\]

\[
\rightarrow (a + c' + d)
\]

\[
\rightarrow (a + c' + d')
\]

\[
\rightarrow (b' + c' + d)
\]

\[
\rightarrow (a' + b + c')
\]

\[
\rightarrow (a' + b' + c)
\]
Basic DLL Search

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(a' + b + c) \\
(a' + b' + c)
\end{align*}
\]

\[
\begin{align*}
\text{Implication Graph} \quad \text{Conflict!}
\end{align*}
\]
Basic DLL Search

\[
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c)
\]

\[
\begin{array}{c}
\Rightarrow \text{Backtrack}
\end{array}
\]
Basic DLL Search

\[
\begin{align*}
(a' + b + c) & \quad \rightarrow \quad (a + c + d) \\
& \quad \rightarrow \quad (a + c + d') \\
& \quad \rightarrow \quad (a + c' + d) \\
& \quad \rightarrow \quad (a + c' + d') \\
\end{align*}
\]

\[
\begin{align*}
(b' + c' + d) & \quad \rightarrow \quad (a' + b + c') \\
& \quad \rightarrow \quad (a' + b' + c) \\
\end{align*}
\]
Basic DLL Search

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

Implication Graph

\[a=1\] \[\rightarrow (a' + b' + c)\] \[\rightarrow c=1\]
\[b=1\]
Basic DLL Search

- $(a' + b + c)$
- $(a + c + d)$
- $(a + c + d')$
- $(a + c' + d)$
- $(a + c' + d')$
- $(b' + c' + d)$
- $(a' + b + c')$
- $(a' + b' + c)$

Implication Graph

- $a=1$
- $b=1$
- $c=1$
- $d=1$

- $c=1, d=1$
Basic DLL Search

\[ (a' + b + c) \]
\[ (a + c + d) \]
\[ (a + c + d') \]
\[ (a + c' + d) \]
\[ (a + c' + d') \]
\[ (b' + c' + d) \]
\[ (a' + b + c') \]
\[ (a' + b' + c) \]

Implication Graph

\[ a=1 \]
\[ b=1 \]
\[ c=1 \]
\[ d=1 \]

\[ \Rightarrow \text{ SAT} \]

\[ c=1, d=1 \]
SAT Solvers: A Condensed History

- **Deductive**
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”

- **Backtrack Search**
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment

- **Conflict Driven Clause Learning [CDCL]**
  - GRASP: Integrate a constraint learning procedure, 1996

- **Locality Based Search**
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation

- **“Pre-processing”**
  - Peephole optimization, e.g. miniSAT, 2005
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

Conflict Driven Learning and Non-chronological Backtracking

\[
\begin{align*}
&x_1 + x_4 \\
&x_1 + x_3' + x_8' \\
&x_1 + x_8 + x_{12} \\
&x_2 + x_{11} \\
&x_{7'} + x_3' + x_9 \\
&x_{7'} + x_8 + x_9' \\
&x_7 + x_8 + x_{10'} \\
&x_7 + x_{10} + x_{12'}
\end{align*}
\]

\[x_1 = 0\]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_{7'} + x_3' + x_9 \]
\[ x_{7'} + x_8 + x_{9'} \]
\[ x_{7'} + x_8 + x_{10'} \]
\[ x_{7} + x_{10} + x_{12'} \]
Conflict Driven Learning and Non-chronological Backtracking

\[
\begin{align*}
&x_1 + x_4 \\
&x_1 + x_3' + x_8' \\
&x_1 + x_8 + x_{12} \\
&x_2 + x_{11} \\
&x_7' + x_3' + x_9 \\
&x_7' + x_8 + x_9' \\
&x_7 + x_8 + x_{10'} \\
&x_7 + x_{10} + x_{12'} \\
\end{align*}
\]

\[x_4 = 1\]  \[x_1 = 0, \ x_4 = 1\]  \[x_3 = 1\]  \[x_1 = 0\]  \[x_3 = 1\]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_{7'} + x_3' + x_9 \]
\[ x_{7'} + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

- \( x_1 = 0, x_4 = 1 \)
- \( x_3 = 1, x_8 = 0 \)
- \( x_8 = 0 \)
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10}' \]
\[ x_7 + x_{10} + x_{12}' \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10}' \]
\[ x_7 + x_{10} + x_{12}' \]

\[ x_3 = 1, \ x_8 = 0, \ x_{12} = 1 \]
\[ x_2 = 0 \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \\
 x_1 + x_3' + x_8' \\
 x_1 + x_8 + x_{12} \\
 x_2 + x_{11} \\
 x_7' + x_3' + x_9 \\
 x_7' + x_8 + x_9' \\
 x_7 + x_8 + x_{10}' \\
 x_7 + x_{10} + x_{12}' \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \\
x_1 + x_3' + x_8' \\
x_1 + x_8 + x_12 \\
x_2 + x_11 \\
x_7' + x_3' + x_9 \\
x_7' + x_8 + x_9' \\
x_7 + x_8 + x_10' \\
x_7 + x_10 + x_12' \]

\[ x_1 = 0, x_4 = 1 \\
x_3 = 1, x_8 = 0, x_12 = 1 \\
x_2 = 0, x_11 = 1 \\
x_7 = 1 \]
Conflict Driven Learning and Non-chronological Backtracking

\[\begin{align*}
x_1 + x_4 \\
x_1 + x_3' + x_8' \\
x_1 + x_8 + x_{12} \\
x_2 + x_{11} \\
x_7' + x_3' + x_9 \\
x_7' + x_8 + x_9' \\
x_7 + x_8 + x_{10'} \\
x_7 + x_{10} + x_{12'} \\
\end{align*}\]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_12 \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

\[ x_1=0, x_4=1 \]
\[ x_3=1, x_8=0, x_12=1 \]
\[ x_2=0, x_{11}=1 \]
\[ x_7=1, x_9=1 \]

\[ x_3=1 \land x_7=1 \land x_1=0 \rightarrow \text{conflict} \]
Conflict Driven Learning and Non-chronological Backtracking

\[ \begin{align*}
&x_1 + x_4 \\
&x_1 + x_3' + x_8' \\
&x_1 + x_8 + x_{12} \\
&x_2 + x_{11} \\
&x_7' + x_3' + x_9 \\
&x_7' + x_8 + x_9' \\
&x_7 + x_8 + x_{10}' \\
&x_7 + x_{10} + x_{12}' \\
\end{align*} \]

\[ \begin{align*}
x_4 = 1 & \quad x_1 = 0, x_4 = 1 \\
x_1 = 0 & \quad x_3 = 1, x_8 = 0, x_{12} = 1 \\
x_3 = 1 & \quad x_2 = 0, x_{11} = 1 \\
x_7 = 1 & \quad x_7 = 1, x_9 = 1 \\
x_9 = 1 & \quad x_1 = 0 \\
x_2 = 0 & \quad x_8 = 0 \\
x_{11} = 1 & \quad x_{12} = 1 \\
\end{align*} \]

\[ x_3 = 1 \land x_7 = 1 \land x_1 = 0 \rightarrow \text{conflict} \]

Add conflict clause: \( x_3' + x_7' + x_1 \)
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

\[ x_3 = 1 \]
\[ \land \]
\[ x_7 = 1 \]
\[ \land \]
\[ x_8 = 0 \]
\[ \rightarrow \text{conflict} \]

\[ x_1 = 0, x_4 = 1 \]
\[ x_2 = 0, x_{11} = 1 \]
\[ x_7 = 1, x_9 = 1 \]

\[ x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict} \]
Conflict Driven Learning and Non-chronological Backtracking

\[ x_1 + x_4 \]
\[ x_1 + x_3' + x_8' \]
\[ x_1 + x_8 + x_{12} \]
\[ x_2 + x_{11} \]
\[ x_7' + x_3' + x_9 \]
\[ x_7' + x_8 + x_9' \]
\[ x_7 + x_8 + x_{10'} \]
\[ x_7 + x_{10} + x_{12'} \]

Add conflict clause: \( x_3' + x_7' + x_8 \)

\( x_1 = 0, x_4 = 1 \)
\( x_3 = 1, x_8 = 0, x_{12} = 1 \)
\( x_2 = 0, x_{11} = 1 \)
\( x_7 = 1, x_9 = 1 \)

\( x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict} \)

Add conflict clause: \( x_3' + x_7' + x_8 \)
Conflict Driven Learning and Non-chronological Backtracking

\[x_1 + x_4\]
\[x_1 + x_3' + x_8'\]
\[x_1 + x_8 + x_{12}\]
\[x_2 + x_{11}\]
\[x_7' + x_3' + x_9\]
\[x_7' + x_8 + x_9'\]
\[x_7 + x_8 + x_{10'}\]
\[x_7 + x_{10} + x_{12'}\]

Add conflict clause: \(x_3' + x_7' + x_8\)

\[x_1 = 0, x_4 = 1\]
\[x_3 = 1, x_8 = 0, x_{12} = 1\]
\[x_2 = 0, x_{11} = 1\]
\[x_7 = 1, x_9 = 1\]

\[x_3 = 1 \land x_7 = 1 \land x_8 = 0 \rightarrow \text{conflict}\]

Add conflict clause: \(x_3' + x_7' + x_8\)
Conflict Driven Learning and Non-chronological Backtracking

- $x_1 + x_4$
- $x_1 + x_3' + x_8'$
- $x_1 + x_8 + x_{12}$
- $x_2 + x_{11}$
- $x_7' + x_3' + x_9$
- $x_7' + x_8 + x_9'$
- $x_7 + x_8 + x_{10'}$
- $x_7 + x_{10} + x_{12'}$

Backtrack to the decision level of $x_3=1$
Conflict Driven Learning and Non-chronological Backtracking

\[
\begin{align*}
x_1 + x_4 \\
x_1 + x_3' + x_8' \\
x_1 + x_8 + x_{12} \\
x_2 + x_{11} \\
x_7' + x_3' + x_9 \\
x_7' + x_8 + x_{9'} \\
x_7 + x_8 + x_{10'} \\
x_7 + x_{10} + x_{12'} \\
x_3' + x_{7'} + x_8
\end{align*}
\]

Backtrack to the decision level of \(x_3 = 1\)
Assign \(x_7 = 0\)
What’s the big deal?

Conflict clause: $x_1' + x_3 + x_5'$

Significantly prune the search space – learned clause is useful forever!

Useful in generating future conflict clauses.
Restart

- Abandon the current search tree and reconstruct a new one
- The clauses learned prior to the restart are still there after the restart and can help pruning the search space
- Adds to robustness in the solver

Conflict clause: $x_1' + x_3 + x_5'$
Conflict Clauses and Resolution

- Conflict clauses can be derived by resolution
- $C_1 = (x_4' + x_2 + x_5)$, $C_2 = (x_4' + x_{10} + x_6)$, $C_3 = (x_5' + x_6' + x_7')$ $C_4 = (x_6' + x_7)$
- $C_5 = \text{Res}(C_4, C_3) = (x_5' + x_6')$
- $C_6 = \text{Res}(C_5, C_2) = (x_4' + x_5' + x_{10})$
- $C_7 = \text{Res}(C_6, C_1) = (x_2 + x_4' + x_{10})$
- $C_7$ is conflict clause
  - only one literal assigned @5
  - clause is “assertive”
  - $x_4$ implied to 0 by unit clause rule at level 3
Algorithm Termination

- Conflict-Driven Clause Learning Algorithm is guaranteed to terminate.
- UNSAT case terminates when the empty clause is learned.
  - Each learned clause is the result of a set of resolutions.

Resolution Proof of Unsatisfiability - can be logged and independently checked
An unsatisfiable core is an inconsistent subset of the clauses of the original formula

\[(\overline{s}) \quad (\overline{r} + s) \quad (r) \quad (s)\]

An unsatisfiable core is minimal if dropping one of its clauses makes it satisfiable.

Can be extracted from resolution proof

Extracting Unsatisfiable Cores

Resolution Proof of Unsatisfiability

- Original Clauses
- Learned Clauses
- Empty Clause

Diagram showing a resolution proof with nodes representing clauses and arrows indicating resolution steps.
Extracting Unsatisfiable Cores

Resolution Proof of Unsatisfiability

- Original Clauses
- Learned Clauses
- Empty Clause
Extracting Unsatisfiable Cores

Resolution Proof of Unsatisfiability

Core Clauses
Original Clauses
Learned Clauses
Empty Clause
SAT Solvers: A Condensed History

- Deductive
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”

- Backtrack Search
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment

- Conflict Driven Clause Learning [CDCL]
  - GRASP: Integrate a constraint learning procedure, 1996

- Locality Based Search
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation

- “Pre-processing”
  - Peephole optimization, e.g. miniSAT, 2005
Success with Chaff

- First major instance: Tough (Industrial Processor Verification)
  - Bounded Model Checking, 14 cycle behavior

- Statistics
  - 1 million variables
  - 10 million literals initially
    - 200 million literals including added clauses
    - 30 million literals finally
  - 4 million clauses (initially)
    - 200K clauses added
  - 1.5 million decisions
  - 3 hour run time

Chaff Contribution 1: Lazy Data Structures
2 Literal Watching for Unit-Propagation

- Avoid expensive book-keeping for unit-propagation
  - Dominates run-time
- N-literal clause can be unit or conflicting only after N-1 of the literals have been assigned to F
  - \((v_1 + v_2 + v_3)\): implied cases: \((0 + 0 + v_3)\) or \((0 + v_2 + 0)\) or \((v_1 + 0 + 0)\)
- Can completely ignore the first N-2 assignments to this clause
- Pick two literals in each clause to “watch” and thus can ignore any assignments to the other literals in the clause.
  - Example: \((v_1 + v_2 + v_3 + v_4 + v_5)\)
  - \((v_1=X + v_2=X + v_3=? \{i.e. X or 0 or 1\} + v_4=? + v_5=? )\)
- Maintain the invariant: If a clause can become newly implied via any sequence of assignments, then this sequence will include an assignment of one of the watched literals to F
For every clause, two literals are watched

- When a variable is assigned true, only need to visit clauses where its watched literal is false (only one polarity)
  - Pointers from each literal to all clauses it is watched in
- In a n clause formula with v variables and m literals
  - Total number of pointers is 2n
  - On average, visit n/v clauses per assignment
- *No updates to watched literals on backtrack*
Decision Heuristics

- Most critical for early search pruning

Conventional Wisdom

- “Assign most tightly constrained variable”: e.g. DLIS (Dynamic Largest Individual Sum)
  - Simple and intuitive: At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - **Expensive book-keeping** operations required
    - Must touch *every* clause that contains a literal that has been set to true. Often restricted to initial (not learned) clauses.
    - Need to reverse the process for un-assignment.

- Look ahead algorithms even more compute intensive
  
  C. Li, Anbulagan, “Look-ahead versus look-back for satisfiability problems”  
Chaff Contribution 2: Activity Based Decision Heuristics

- **VSIDS: Variable State Independent Decaying Sum**
  - Rank variables by literal count in the initial clause database
  - Only increment counts as new (learnt) clauses are added
  - Periodically, divide all counts by a constant

- **Quasi-static:**
  - Static because it doesn’t depend on variable state
  - Not static because it gradually changes as new clauses are added
    - *Decay causes bias toward *recent* conflicts.*
    - Has a beneficial interaction with 2-literal watching
Activity Based Heuristics and Locality Based Search

By focusing on a sub-space, the covered spaces tend to coalesce

- Variable activity based heuristics lead to locality based search
- More opportunities for resolution since most of the variables are common.
SAT Solvers: A Condensed History

- **Deductive**
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”

- **Backtrack Search**
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment

- **Conflict Driven Clause Learning [CDCL]**
  - GRASP: Integrate a constraint learning procedure, 1996

- **Locality Based Search**
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation

- **“Pre-processing”**
  - Peephole optimization, e.g. miniSAT, 2005
Pre-Processing of CNF Formulas

N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination, In Proceedings of SAT 2005

- Use structural information to simplify
  - Subsumption
  - Self-subsumption
  - Substitution
Pre-Processing: Subsumption

- Clause \( C_1 \) subsumes clause \( C_2 \) if \( C_1 \) implies \( C_2 \)
- Subsumed clauses can be discarded

\[
(\overline{x} + y) \cdot (\overline{x} + y + z) \cdot (y + v) \cdot (\overline{x} + v + z + v) \cdot (\overline{y})
\]
Pre-Processing: Self-Subsumption

- Subsumption *after* resolution step

\[
(\overline{x} + y + z) \quad (x + y + z + \overline{u})
\]

\[
(y + z + \overline{u})
\]
Pre-Processing: Substitution

- Tseitin transformation introduces *definition* of variable

\[ y \rightarrow z \rightarrow x_1 \]

\[
\begin{align*}
(x_1 \leftrightarrow (y \leftrightarrow z)) \\
(\overline{x_1} + \overline{y} + z) \cdot (\overline{x_1} + \overline{z} + y) \cdot (\overline{y} + \overline{z} + x_1) \cdot (y + z + x_1)
\end{align*}
\]

- Occurrence of \( x_1 \) can be eliminated by substitution
  - Equivalent to existential abstraction

\[
\begin{align*}
(x_1 + u) \cdot (\overline{x_1} \cdot (y \leftrightarrow z) \cdot (\overline{x_1} + \overline{z} + y)
\end{align*}
\]

\[
\begin{align*}
(u + \overline{y} + z) \cdot (u + x_1) \cdot (u + \overline{z} + y)
\end{align*}
\]
Pre-Processing: Substitution

- In practice: “Definitional” clauses are not known
- Not all substitutions desirable
- Limit to beneficial substitutions
  - Partition clauses:
    \[ S_x = \{ C \mid x \text{ is in } C \}, \quad S_{x'} = \{ C \mid x' \text{ is in } C \} \]
  - Compute all resolvents for \( S_x \) and \( S_{x'} \).
  - If number of resolvents smaller than union of \( S_x \) and \( S_{x'} \), then replace \( S_x \) and \( S_{x'} \) by resolvents
SAT Solvers: Extensions

- All-SAT
  - Iterative enumeration of all solutions
- Cardinality Constraints
  - “Bit-Flattening”
  - Sorting Networks
- Maximum Satisfiability Problem
  - MAX-SAT
  - Partial MAX-SAT
- Minimal Correction Sets
  - Computing Minimal Correction Sets
  - Minimal Correction sets and Unsatisfiable Cores
All-SAT

- Some applications require enumeration of all satisfying assignments
  - E.g., predicate abstraction (Boolean abstraction)
- Complexity class #P
  - # satisfying assignments of propositional formula
- Implementation: Incremental SAT
Incremental All-SAT

- Find satisfying assignment
  - Corresponds to conjunction of literals
    - E.g.: $x_2=0 \land x_3=0 \land x_1=1 \land x_5=1$

- Add negation of assignment to formula
  - E.g. $(x_2 + x_3 + x_1' + x_5')$
    - *blocking clause*
    - Eliminates current assignment

- Repeat until formula becomes unsatisfiable
  - Incremental: Learnt clauses are preserved across iterations

- Use partial satisfying assignments to cover entire subspaces
  - E.g. $x_1=1$ covers half the space
Incremental All-SAT: Example

\[(x_2 + x_3) (x_1' + x_4) (x_2' + x_4) (x_1' + x_2 + x_3') (x_1') (x_1 + x_2 + x_3')\]

1.) \(x_1 = 0, x_2 = 0, x_3 = 1\)  
2.) \(x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 1\)
SAT Solvers: Extensions

- All-SAT
  - Iterative enumeration of all solutions
- Cardinality Constraints
  - “Bit-Flattening”
  - Sorting Networks
- Maximum Satisfiability Problem
  - MAX-SAT
  - Partial MAX-SAT
- Minimal Correction Sets
  - Computing Minimal Correction Sets
  - Minimal Correction sets and Unsatisfiable Cores
Cardinality Constraints

- Linear inequalities over variables encoded in SAT
  - “At most k variables evaluate to 1”
- Obvious encoding

\[
\sum_{i=1}^{n} x_i \leq k
\]

Poor performance 😞
Cardinality Constraints

- Alternate encoding with better unit-propagation
  - Using sorting networks

<table>
<thead>
<tr>
<th></th>
<th>i₁</th>
<th>i₂</th>
<th>o₁</th>
<th>o₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

o₁ = i₁ + i₂
o₂ = i₁ • i₂
Cardinality Constraints

- Larger sorting networks by means of cascading
- Uses only AND-gates and OR-gates
Cardinality Constraints

- “At most 2 variables in \((i_1, i_2, i_3, i_4)\) are assigned 1”

Very good performance! Some understanding of the internals of the SAT Solver pays off 😊
SAT Solvers: Extensions

- All-SAT
  - Iterative enumeration of all solutions
- Cardinality Constraints
  - “Bit-Flattening”
  - Sorting Networks
- Maximum Satisfiability Problem
  - MAX-SAT
  - Partial MAX-SAT
- Minimal Correction Sets
  - Computing Minimal Correction Sets
  - Minimal Correction sets and Unsatisfiable Cores
**MAX SAT**

- Maximum number of clauses that can be satisfied
- Given an **UNSAT** instance

$$(\bar{r} + \bar{s} + t) \quad (\bar{r} + s) \quad (r) \quad (s) \quad (\bar{t})$$

- What is the largest subset of clauses that can be satisfied?
- Or equivalently:
- What is the smallest subset of clauses that must be dropped to make it satisfiable?
Partial MAX SAT

- Maximum number of clauses that can be satisfied given certain clauses can’t be dropped
  - Correspond to hard constraints

- Given an UNSAT instance

\[(\overline{r} + \overline{s} + t) (\overline{r} + s) (\overline{r}_j) (s) (\overline{t}_j)\]

Which clauses do we have to drop to make it satisfiable?
(the pinned clauses can’t be dropped)
Partial MAX-SAT

“Dropping clauses” in SAT solvers:

- Introduce *relaxation variables* as a “switch”:

  Given: \((x + y')\)

  Relaxed clause \((r + x + y')\)

\[
\begin{align*}
  r &= 0 & r &= 1 \\
  \text{ON} & & \text{OFF}
\end{align*}
\]
Core-Guided MAX-SAT

- **Observation:**
  - Dropped clauses must deactivate all UNSAT cores
    - Clauses not contained in any core need not be dropped

\[(r + t) \cdot (r + s) \cdot (s) \cdot (\overline{s}) \cdot (t) \cdot (\overline{t})\]

- **Idea:**
  - Find a core
  - Instrument its clauses with relaxation literals
  - Use cardinality constraints to restrict # of clauses dropped
Core-Guided MAX-SAT

- Find an initial core
- Instrument it
- Add cardinality constraint
- Search for next core
- ...
- Satisfying assignment indicates dropped clauses
  - $x=0, y=1, u=0, v=1$
  - $(r + t) (r + s) (x(s^t) s) ((s^t) s') (u(t^v) t) (v(t^v)t') (x' + y') (u' + v')$
Core-Guided Partial MAX-SAT

- The assignment $x=0$, $y=1$, $u=0$, $v=1$ tells us which clauses to drop

\[(r + t) \cdot (r + s) \cdot (s) \cdot (\bar{s}) \cdot (t) \cdot (\bar{t})\]

- Dropping $(s')$ and $(t')$ “corrects” the formula
  - $\{(s'), (t')\}$ is a “minimal correction set” (MCS)

- Algorithm can be easily extended to Partial MAX-SAT
  - “Hard” clauses cannot be dropped
  - Never instrument hard clauses
SAT Solvers: Extensions

- All-SAT
  - Iterative enumeration of all solutions
- Cardinality Constraints
  - “Bit-Flattening”
  - Sorting Networks
- Maximum Satisfiability Problem
  - MAX-SAT
  - Partial MAX-SAT
- Minimal Correction Sets
  - Computing Minimal Correction Sets
  - Minimal Correction sets and Unsatisfiable Cores
Minimal Correction Set (MCS)

- **Minimal Correction Set:**
  - Minimal set of clauses which need to be dropped to make formula satisfiable

- The complement of any MAX-SAT solution is an MCS

- Converse doesn’t hold:
  Complement of MCS is maximal set of satisfiable clauses
Minimal Correction Set (MCS)

- The following formula has a single minimum MCS:

$$(\overline{s}), (\overline{r} + s), (r), (s)$$

- $\{(r),(s)\}$ is minimal but not minimum.

- The formula has three different MCSes.
Computing Minimal Correction Sets

- Based on Core-Guided MAX-SAT
- MAX-SAT algorithm already provides first MCS
- Idea:
  - Use All-SAT to enumerate all MCSes
  - Blocking clauses eliminate already discovered MCSes
Computing Minimal Correction Sets

- Find first MCS using MAX-SAT
- Block this solution by blocking relaxation literal

\[(x') \quad (x(s's')) \quad (r' + s) \quad (y(s)s) \quad (r) \quad (x' + y')\]
Computing MCSes (contd.)

- Find the second core

\[(x') (x + s') (r' + s) (y + s) (r)\]

- Add new relaxation literals to respective clauses

- Add cardinality constraint

\[(x') (x + s') (z + r' + s) (y + s) (u + r) \sum (x, z, y, u) \leq 2\]

- Enumerate all MCSes of size 2
Computing MCSes (contd.)

- Block the two newly discovered MCSes
  - Resulting formula is unsatisfiable
  - No more MCSes of size $\leq 2$

\[
(z' + y') (y' + u') (x') (x + s') (z + r' + s) (y + s) (u + r)
\]

\[
\left\{ s, (r + s) \right\} \quad \left\{ r, s \right\} \quad \left\{ \overline{s} \right\}
\]
Unsatisfiable Cores and MCSes

- Dropping the clauses of an MCS “deactivates” all cores
  - Each minimal hitting set of all cores is an MCS

### Cores

- \{ (\overline{s}), (s) \}
- \{ (\overline{s}), (r), (\overline{r} + s) \}

### MCSes

- \{ (\overline{s}) \}
- \{ (r), (s) \}
- \{ (s), (\overline{r} + s) \}
MCSes and Unsatisfiable Cores

- Generate all MCSes for a set of clauses

\[ (\bar{s}) \quad (\bar{r} + s) \quad (r) \quad (s) \]

- Each minimal hitting set of the MCSes is a minimal UNSAT core
Computing Hitting Sets with SAT

- Encode hitting set as a SAT instance (montone)
  - E.g. \{x_1,x_2\}, \{x_2,x_3,x_4\}, \{x_3,x_5\} encoded as:
    - \((x_1+x_2)(x_2+x_3+x_4)(x_3+x_5)\)

- Each satisfying solution is a hitting set

- All satisfying solutions give all hitting sets
  - Block hitting sets already seen

- Generate solutions in increasing size using cardinality constraints to get minimal hitting sets
  - \((x_1+x_2)(x_2+x_3+x_4)(x_3+x_5) \sum x_i \leq k\)
  - \{x_1,x_3\} will be generated first and block \{x_1,x_3,x_4\}, \{x_1,x_2, x_3\} (and others) from being generated
SAT: Techniques and Extensions

- Part I: SAT Solving Techniques
- Part II: Extensions
- Part III: Example Application
  - Design Debug/Fault Localization/Fault Diagnosis
Test Case Generation

- Assumption: We’re given a “golden model”
- We want test cases to check whether chip behaves correctly
Test Case Generation

- Any satisfying assignment of unwound encoding of the model is a test scenario

\[
\begin{align*}
\text{cycle 1:} & \quad (\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (\overline{i_1^1} + \overline{s} + r) \\
\text{cycle 2:} & \quad (\overline{t} + i_2^2) \cdot (\overline{t} + r) \cdot (\overline{i_2^2} + \overline{r} + t)
\end{align*}
\]

- Test Harness

- \( s=0, i_1=1, i_2=0, o=0, r=0 \)
- \( i_1=1, i_2=0, o=0, t=0 \)
Test Case Generation

- Resulting test scenario used to test manufactured prototype

```
s=0, i_1=1, i_2=0, o=0
i_1=1, i_2=0, o=0, t=0
```

Failing test gives a counterexample
BMC and Property Falsification

- **Bounded Model Checking**
  Check whether given property fails within $k$ cycles

```
\[ (\overline{s} \cdot \overline{i_1^1}) \rightarrow \overline{o_1} \]
```

```
\[ (\overline{s} \cdot \overline{i_1^1} \cdot \overline{i_1^2}) \rightarrow \overline{o_2} \]
```

“output remains 0 as long as initial state ($s$) and input 1 ($i_1$) are 0”
BMC and Property Falsification

- Property fails if unfolding and negated property is \textbf{SAT}
- Any satisfying assignment is a \texttt{counterexample} to the claim

\[
\begin{align*}
\text{cycle 1} & \quad (\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (\overline{i_1^1} + \overline{s} + r) \\
\text{cycle 2} & \quad (\overline{t} + i_1^2) \cdot (\overline{t} + r) \cdot (\overline{i_1^2} + \overline{r} + t)
\end{align*}
\]

\[
\begin{align*}
\left(\overline{s} \cdot \overline{i_1^1}\right) & \rightarrow \overline{o^1} \\
\left(\overline{s} \cdot \overline{i_1^1} \cdot \overline{i_1^2}\right) & \rightarrow \overline{o^2}
\end{align*}
\]
BMC and Property Falsification

Any satisfying assignment is a counterexample to the claim

$$\left( \overline{s} \cdot i_1^1 \right) \rightarrow \overline{o^1}$$
Counterexamples

- More valuable than correctness proofs?

  “One of the most important advantages of model checking [...] is its counterexample facility. [...] The counterexamples can be essential in finding subtle errors in designs.”

  [Clarke, Grumberg, McMillan, Zhao 1995]

- However, a counterexample still doesn’t tell us
  - what went wrong, and
  - where it went wrong.

  Root Cause Analysis or Fault Localization
Unsatisfiable Instances

- Constrain a faulty design with the correct input/output
- The resulting formula is unsatisfiable
- Locate gates that are inconsistent with desired behavior
SAT Based Design Debug

Golden Model

Specification

Unfolding

Logic net-list

Derive

Requirements, UML, Use Cases, SystemC, ...

Test Scenario

Consistent?

Iterative Logic Array
Test Case as Circuit Constraints

- Test scenario is modeled as constraint for iterative logic array
Test Scenarios as Constraints

**Specification**

“output remains 0 as long as initial state (s) and input 1 (i₁) are 0”

**Logic net-list**

<table>
<thead>
<tr>
<th>Time-frame</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>i₁</td>
</tr>
<tr>
<td>i₁</td>
<td>0</td>
<td>i₂</td>
</tr>
<tr>
<td>i₂</td>
<td>0</td>
<td>o</td>
</tr>
<tr>
<td>o</td>
<td>0</td>
<td>t</td>
</tr>
</tbody>
</table>

Time-frame 1:
- \( s = 0 \)
- \( i₁ = 0 \)
- \( i₂ = 0 \)
- \( o = 0 \)
- \( t = 0 \)

Time-frame 2:
- \( s = 0 \)
- \( i₁ = 0 \)
- \( i₂ = 1 \)
- \( o = 0 \)
- \( t = 0 \)
Test Scenarios as Constraints

- Add test-scenario as constraints to circuit
- The corresponding CNF formula is inconsistent
Test Scenarios as Constraints

- *Detecting* the error is only half the story
- Manually *localizing* the fault causing a known error is tedious
Fault Localization Using MCSes

- Use MCSes to identify error location.
- Input/output values are hard constraints (we’re not interested in MCSes including them).

\[
(s) \ (i_1^1) \ (i_2^1) \ (o^1) \quad (t) \ (i_1^2) \ (i_2^2) \ (o^2)
\]

\[
\begin{align*}
\text{cycle } 1: \quad & (\overline{r} + i_1^1) \cdot (\overline{r} + s) \cdot (i_1^1 + \overline{s} + r) \\
\text{cycle } 2: \quad & (t + i_2^2) \cdot (\overline{t} + r) \cdot (i_1^2 + \overline{r} + t)
\end{align*}
\]
Fault Localization Using MCSes

- Generate ILA constrained with test case (in CNF)
- Compute all MCSes:
  Each MCS represents a set of potential fault locations
Fault Localization in Software

- **Methodology**
  1. Observe an assertion violation
  2. Unwind loops of the program, obtain symbolic representation
  3. Constrain program with expected input/output values
  4. Compute MCSes to locate faults

M. Jose, R. Majumdar, Cause Cue Clauses: Error Localization Using Maximum Satisfiability. *Programming Languages Design and Implementation*, 2011
Fault Localization in Software

Problem Definition

Golden Model

 Specification
  e.g., requirements

Software Implementation

Assertions in Program

Consistent?

Symbolic Representation

Unwinding
Symbolic Representation of Software

$\text{guard}_1 = (\text{index}_1 \neq 1)$

$\text{index}_2 = 2$

$\text{index}_3 = \text{index}_1 + 2$

$i = \text{guard}_1 ? \text{index}_2 : \text{index}_3$

Static Single Assignment Form [Cytron, Ferrante, Rosen, Wegman, Zadeck 1991]
Computing MCSes for Program

```plaintext
index_1 = 1

guard_1 = (index_1 ≠ 1)

index_2 = 2

index_3 = index_1 + 2

i = guard_1? index_2 : index_3

(i < 3)
```

test input

violated assertion
MCSes Indicate Potential Errors

```c
int Array[3];

int testme(int index)
{
    if (index != 1)
        index = 2;
    else
        index = index + 2;
    i = index;

    assert (i >= 0 && i < 3); // array bounds
    return Array[i];
}
```
Debugging Hardware Prototypes

- Long traces
  - Long separation between fault excitation and error observation
- Limited observability of signals in manufactured chip
  - Trace buffers: Limited recording of select signals
  - Scan chains: Read-out after chip execution stopped

![Diagram of hardware prototypes with trace buffer and scan chain connections]
Hardware Faults and MCSes

- Limited observability results in harder decision problems
- Analysing ILA with millions of time-frames becomes computationally infeasible
Hardware Faults and MCSes

- Analysis limited to small (contiguous) sequence of cycles
- Scalability of decision procedure determines window size
- Slide window backwards in time to cover different cycles
Sliding Windows

- Example

- Sliding windows may fail to locate fault
- Approach is incomplete due to limited information
  - In this particular example: we don’t know the value of r
Example

Would like to propagate information across windows
At a reasonable computational cost
Maybe we can infer the value of $r$ in the first window?
Reconstructing Information

With Inferred Values

\[ r = 1 \]
Inferring “Fixed” Signals for Satisfiable Instances

- Backbone of a satisfiable formula:
  Set of variables that have *same* value in all satisfying assignments

- Consider the satisfiable formula

\[(r \oplus t) \cdot (r + s) \cdot (r)\]

- Satisfying assignments:

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Computing Backbones

Given a Boolean formula $F$
1. Obtain initial satisfying assignment $A_0$
2. For each literal $p$ such that $A_0[p] = 1$
   - variable of $p$ is part of backbone iff $(F \cdot p')$ is UNSAT

Optimization (Filtering):
1. If $(F \cdot p')$ is satisfiable, look at this satisfying assignment $A_1$
2. Variables differing in value in $A_0$ and $A_1$ are not in backbone

Propagating Backbones

- Across Sliding Windows
  - Backbones provide a specific abstraction

Charlie Shucheng Zhu, Georg Weissenbacher and Sharad Malik,
Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones,
Concluding Remarks

- **SAT**: Significant shift from theoretical interest to practical impact.
- Quantum leaps between generations of SAT solvers
- Successful application of diverse CS techniques
  - Logic (Deduction and Solving), Search, Caching, Randomization, Data structures, efficient algorithms
  - Engineering developments through experimental computer science
- Presence of drivers results in maximum progress.
  - Electronic design automation – primary driver and main beneficiary
  - Software verification - the next frontier
- Opens attack on even harder problems
  - SMT, Max-SAT, QBF…

References

References

- [LS209] Liffiton, Sakallah, Generalizing Core-Guided MAX-SAT, Theory and Applications of Satisfiability Testing (SAT) 2009
References

Come to Princeton

- Give a seminar
- Post-doc/PhD positions