SAT-based Approaches for Test & Verification of Integrated Circuits

Dr. Tobias Schubert
Chair of Computer Architecture
Institute of Computer Science
Faculty of Engineering
schubert@informatik.uni-freiburg.de

Summer School on Verification Technology, Systems & Applications 2015
About Me

Just a very short CV

- Studied computer science & microsystems engineering at the University of Freiburg
- Made my PhD working on efficient parallel SAT solving at the University of Freiburg
- Member of the *Transregional Collaborative Research Center 14 AVACS – Automatic Verification and Analysis of Complex Systems*
- Principal investigator within the cluster of excellence *BrainLinks-BrainTools*
- Member of the part-time distance learning program *Intelligent Embedded Microsystems*
My research interests include

- Efficient (parallel) algorithms for SAT and related domains
- Real-world applications using
  - SAT,
  - #SAT,
  - MaxSAT,
  - QBF, and
  - SMT solvers
    as the underlying backend
- Embedded & cyber-physical systems
- Industrial internet & internet of things
- E-learning, blended learning, distance teaching
Collaborators

University of Freiburg
- Bernd Becker
- Jan Burchard
- Alejandro Czutro
- Linus Feiten
- Karina Gitina
- Paolo Marin
- Sven Reimer
- Matthias Sauer
- Karsten Scheibler
- Christoph Scholl
- Ralf Wimmer

University of Bremen
- Rolf Drechsler

University of Oldenburg
- Martin Fränzle

University of Passau
- Ilia Polian

University of Potsdam
- Torsten Schaub

MPI Saarbrücken
- Christoph Weidenbach
Motivation: Embedded Systems

Embedded Systems

- Information processing systems embedded into a “larger” product

Without Embedded Systems

- No cars would drive today
- No planes would fly today
- No factory would work today
- No mobile communication would be possible
Motivation: Embedded Systems

Embedded Systems

- Information processing systems embedded into a “larger” product

Without Embedded Systems

- No cars would drive today
- No planes would fly today
- No factory would work today
- No mobile communication would be possible

Verifying designs and testing produced chips are mandatory tasks, in particular for safety-critical applications!
Motivation: Automotive Area

- Many functions controlled by embedded systems
- Multiple networks / system busses
- Up to 70 different processors within one car
Motivation: Automotive Area

Consequences

- Increasing system complexity
- Increasing number of dependencies between different subsystems
- Up to 40% of the total costs are caused by electronics & software
- Up to 90% of the innovations are driven by electronics & software
- 40–50% of all car breakdowns are caused by electronics & software
- Errors related to electronics or software are responsible for more than 40% of all call-backs
- Reliable function is of outmost importance, because otherwise human lives can be endangered!

⇒ Safety-critical application of embedded systems!
Verifying Integrated Circuit Designs

Focus is on detecting design errors

- Errors which occur during the translation of a specification into the final integrated circuit (implementation)
- Errors in the design make all produced chips erroneous

⇒ Formal methods to avoid design errors before producing any chip
Testing Integrated Circuits

Focus is on production errors

- Defects which are caused during the production of single chips and which change their functionality
- Causes are contaminations, shifted exposure masks, wrong doping, ...

⇒ Formal methods to ensure that all production errors can be found
But why using SAT Solvers?

- Tremendous performance improvements within the last 15 years
- Nowadays SAT solvers (and their extensions) are able to …
  - solve problems coming from real-world applications (e.g., large industrial circuits)
  - handle optimization & enumeration problems, multi-valued domains, hybrid systems

\[ 0, 1 \times \]
Typical SAT-based Flow

"real problem"

(a + b + c) · (a + c) · 
(b + c) · (c + d) · 
(c + d) · (d + e + f) · 
(d + f) · (e + f) · f

SAT instance

SAT solver

SAT solution

\[ a = 0, b = 0, \]
\[ c = 0, d = 1, \]
\[ e = 1, f = 1 \]
Outline

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT

VTSA’15 Tobias Schubert – SAT-based Test & Verification 12 / 192
Outline

Applications

Bounded Model / Property Checking
Path Compaction
Security Issues
Test Pattern Relaxation
Automatic Test Pattern Generation
Hybrid System Verification
Black Box Verification
Combinational Equivalence Checking
The End

Core Algorithms

SAT
MaxSAT
#SAT
QBF
DQBF
SMT
Boolean Satisfiability Problem (SAT)

- **Given**
  - A Boolean formula \( \varphi \) in Conjunctive Normal Form (CNF)
    - A CNF is a conjunction of clauses: \( C_1 \land \ldots \land C_m \)
    - A clause is a disjunction of literals: \( (l_1 \lor \ldots \lor l_k) \)
    - A literal \( l \) is a Boolean variable or its negation: \( l \) or \( \neg l \)

- **Question**
  - Is there a valuation of the variables that satisfies \( \varphi \)?

- **Example**
  - \( x_1 = x_2 = 0 \), \( x_3 = 1 \) satisfies
  - \( \varphi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \)

- Techniques for solving instances of the SAT problem are called SAT algorithms or SAT solvers

- Complexity of the “general” SAT problem: NP-complete (S.A. Cook, 1971)
Overview of SAT Algorithms

Focus here is on complete methods

- Due to a systematic procedure complete solvers are able to prove the unsatisfiability of a CNF formula

- **DP algorithm**
  - M. Davis, H. Putnam, 1960
  - Based on resolution

- **DLL algorithm**
  - M. Davis, G. Logemann, D. Loveland, 1962
  - Based on depth-first search

- **Modern SAT algorithms**
  - Based on the DLL algorithm, but enriched with efficient data structures and several acceleration & optimization techniques
  - zChaff, MiniSat, MiraXT, lingeling, antom, Glucose
Preliminaries

Definition (Empty Clause)

The empty clause, denoted with $\Box$, describes the empty set of literals, and it is unsatisfiable by definition.

Definition (Empty Formula)

The empty formula describes an empty set of clauses and it is satisfiable by definition.
Preliminaries

Definition (Pure Literal)

Let $F$ be a CNF formula and $L$ be a literal contained in $F$. $L$ is called a pure literal iff $L$ occurs in $F$ only positive or only negative.

Steps in order to simplify a CNF formula $F$

- Delete from $F$ all clauses in which a pure literal $L$ occurs, because these ones will be satisfied by an appropriate assignment to $L$.

Remark

- As it is rather time consuming, pure literal detection is applied by modern SAT solvers during pre-/inprocessing only.
Preliminaries

Definition (Unit Clause)

A clause consisting of a single literal $L$ is called a unit clause with $L$ being the corresponding unit literal.

Steps in order to simplify a CNF formula $F$

- Assign a unit literal $L$ to 1
- Delete from $F$ all clauses containing $L$
- Delete all occurrences of $\neg L$

$$F = (\neg x) \land (a \lor b) \land (x \lor y) \land \neg \neg x$$
Definition (Subsumption)

Let $C_1$ and $C_2$ be two clauses. $C_1$ subsumes $C_2$ iff all literals occurring in $C_1$ also occur in $C_2$: $C_1 \subseteq C_2$.

Steps in order to simplify a CNF formula $F$

- Delete all clauses from $F$ that are subsumed by at least one other clause of $F$

Remark

- Typically, modern SAT solvers apply subsumption checks during pre-/inprocessing only
**Definition (Resolution)**

Let $C_1$ and $C_2$ be two clauses and $L$ be a literal with the following property: $L \in C_1$ and $\neg L \in C_2$. Then one can compute the clause $R$:

$$R = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

that is denoted as the resolvent of the clauses $C_1$ and $C_2$ over $L$. Typically, the notation $R = C_1 \otimes_L C_2$ is used.

**Lemma (Resolution Lemma)**

*Let $F$ be a CNF formula and $R$ be the resolvent of two clauses $C_1$ and $C_2$ from $F$. Then $F$ and $F \cup \{R\}$ are equivalent: $F \equiv F \cup \{R\}$.*
Preliminaries

Definition

Let $F$ be a CNF formula. Then $\text{Res}(F)$ is defined as

$$\text{Res}(F) = F \cup \{R \mid R \text{ is the resolvent of two clauses in } F\}.$$ 

Moreover, let us define:

$$\text{Res}^0(F) = F$$

$$\text{Res}^{t+1}(F) = \text{Res}(\text{Res}^t(F)) \text{ for } t \geq 0$$

$$\text{Res}^*(F) = \lim_{t \to \infty} \text{Res}^t(F)$$

Theorem (Resolution Theorem)

A CNF formula $F$ is unsatisfiable iff $\Box \in \text{Res}^*(F)$.
Let $F$ be a CNF formula and $x_i$ a variable occurring in $F$ with $L = x_i$ and $\neg L = \neg x_i$. The we define $P$, $N$ and $W$ as follows:

- $P$ is the set of clauses in $F$ which contain $L$:
  $$P = \{ C \in F \mid L \in C \}$$

- $N$ is the set of clauses in $F$ which contain $\neg L$:
  $$N = \{ C \in F \mid \neg L \in C \}$$

- $W$ is the set of clauses in $F$ which contain neither $L$ nor $\neg L$:
  $$W = \{ C \in F \mid L \notin C \land \neg L \notin C \}$$

Obviously, we have $F = P \cup N \cup W$. 
Definition (Pairwise Resolution)

Using this partitioning of the clauses we define $P \otimes_{x_i} N$ as the set of clauses, which can be constructed by resolution of all pairs $(p, n) \in P \times N$:

$$P \otimes_{x_i} N = \{ R \mid (R = C_1 \otimes_{x_i} C_2) \wedge (C_1 \in P) \wedge (C_2 \in N) \}.$$ 

Theorem (Variable Elimination)

Let $F$ be a formula in CNF and $x_i$ a variable which appears both positive and negative in $F$. Further let the sets $P$, $N$, and $W$ be the partition of $F$ as defined before.

Then $F = P \cup N \cup W$ and $F' = (P \otimes_{x_i} N) \wedge W$ are satisfiability equivalent.
DLL Algorithm

- Main idea: If a CNF formula $F$ is satisfiable, then for an arbitrary variable $x_i$ occurring in $F$ either $x_i = 1$ or $x_i = 0$ must hold
  ⇒ Try both cases one after the other
  ⇒ Depth-first search

- Applying unit clause & pure literal rule to accelerate the search

- Recursive algorithm, in particular the given formula gets modified when going from recursion level $r$ to $r + 1$

- In the literature both “DLL” and “DPLL” can be found
bool DLL(CNF F)
{
    if (F == ∅) { return SATISFIABLE; } // Empty set of clauses
    if (F[0] ∈ F) { return UNSATISFIABLE; } // Empty Clause
    if (F contains a unit clause (L)) // Unit Clause
    {
        // Unit Subsumption.
        F' = F - {C | (L ∈ C) ∧ (C ∈ F) ∧ (C ≠ (L))};
        // Unit Resolution.
        P = {L};
        N = {C | (¬L ∈ C) ∧ (C ∈ F')};
        W = F' - P - N;
        return DLL([P ⊗ L N] ∧ W);
    }
    if (F contains a pure literal L) // Pure Literal
    {
        // Delete from F every clause containing L.
        F' = F - {C | (L ∈ C) ∧ (C ∈ F)};
        return DLL(F');
    }
    L = SELECTLITERAL(F);
    if (DLL(F ∪ {L}) == SATISFIABLE) // Case distinction
    { return SATISFIABLE; }
    else
    { return DLL(F ∪ {¬L}); }
}
DLL Algorithm

\[ (\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3) \]
DLL Algorithm

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]

Case distinction \(x_1 = 1\)
DLL Algorithm

\[ (\neg x_2, \neg x_3) \land (\neg x_2, x_3) \land (x_2, \neg x_3) \land (x_2, x_3) \land \]

Case distinction \( x_1 = 1 \)
DLL Algorithm

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_2, x_3) \land (x_2, \neg x_3) \land (x_1, \neg x_2, \neg x_3) \land (x_1, x_2, x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, x_3) \land (x_1, x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (x_1, \neg x_2, \neg x_3) \land (x_1, x_2, x_3)]

Case distinction $x_2 = 1$
Case distinction $x_2 = 1$
DLL Algorithm

\((\neg x_3) \land (x_3)\)

Unit clauses $x_3 = 0$ and $x_3 = 1$
DLL Algorithm

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)]

Contradiction/conflict
DLL Algorithm

\[
\begin{align*}
\neg x_1, \neg x_2, \neg x_3 & \land \neg x_1, x_2, \neg x_3 \\
\neg x_1, x_2, x_3 & \land x_1, \neg x_2, \neg x_3 \\
\neg x_1, \neg x_2, x_3 & \land x_1, x_2, x_3
\end{align*}
\]

Case distinction \(x_2 = 0\)
DLL Algorithm

\[ \neg x_1, \neg x_2, \neg x_3 \land \neg x_1, \neg x_2, x_3 \land \neg x_1, x_2, \neg x_3 \land x_1, \neg x_2, \neg x_3 \]

Case distinction $x_2 = 0$
Unit clauses $x_3 = 0$ and $x_3 = 1$
\[
\begin{align*}
\land & \land (\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land \\
\text{Contradiction/conflict}
\end{align*}
\]
DLL Algorithm

\[ (\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3) \]

Case distinction \( x_1 = 0 \)
DLL Algorithm

\[ (\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (x_1, \neg x_2, \neg x_3) \]

Case distinction \( x_1 = 0 \)
DLL Algorithm

\[\neg x_1, \neg x_2, \neg x_3 \wedge \neg x_1, \neg x_2, x_3 \wedge \neg x_1, x_2, \neg x_3 \wedge \neg x_1, x_2, x_3 \wedge (x_1, \neg x_2, \neg x_3)\]

Pure literal \( x_2 = 0 \)
DLL Algorithm

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]

Pure literal \(x_2 = 0\)
DLL Algorithm

\[
\begin{align*}
(x_1 & \lor \neg x_2 \land \neg x_3) \\
\land (\neg x_1 & \lor \neg x_2 \land x_3) \\
\land (\neg x_1 & \lor x_2 \land \neg x_3) \\
\land (x_1 & \lor \neg x_2 \land \neg x_3)
\end{align*}
\]

Formula satisfiable
From DLL to modern SAT Algorithms

Overall

- DLL algorithm
  - Recursive procedure
  - For the transition from recursion level $r$ to level $r + 1$ the given formula gets modified
  - For backtracking from level $r + 1$ to $r$ the original (sub)formula at level $r$ has to be restored

- Modern SAT algorithms
  - Non-recursive implementation
  - Apart from special cases (preprocessing), the CNF remains unmodified
  - Typically, the pure literal rule is not applied
From DLL to modern SAT Algorithms

**Unit clause**

- DLL algorithm
- A clause consisting exactly one literal
- Modern SAT algorithms

- In addition to the rule above, clauses where all literals but one are assigned with negated polarity are also referred to as unit clauses
- Example: Assignment $x_1 = 0, x_2 = 1$ turns $(x_1, \neg x_2, x_3)$ into a unit clause
- In the example, the evaluation $x_1 = 0, x_2 = 1$ forces the assignment $x_3 = 1$ in order to satisfy the clause $(x_1, \neg x_2, x_3)$

$\Rightarrow$ implication

$x_1 = 0, x_2 = 1 \implies x_3 = 1$
From DLL to modern SAT Algorithms

Unit propagation to determine all implications forced by a variable assignment

- DLL algorithm
  - Repeated application of the unit clause rule on successive recursion levels until the rule cannot be applied anymore

- Modern SAT algorithms
  - Done non-recursively, also called **Boolean Constraint Propagation (BCP)**
  - Example: For the CNF $F = (x_1, \lnot x_2) \land (x_1, x_2, x_3) \land (\lnot x_3, x_4)$,
    $x_1 = 0$ leads to the implications $x_2 = 0, x_3 = 1, x_4 = 1$
From DLL to modern SAT Algorithms

Contradiction/conflict

- DLL algorithm
  - Empty clause
- Modern SAT algorithms
  - Unsatisfied clause
  - Example: Valuation $x_1 = 0, x_2 = 1, x_3 = 0$ makes $(x_1, \neg x_2, x_3)$ unsatisfied, and so the whole CNF formula containing it cannot be satisfied anymore
Conflict analysis & backtracking

- DLL algorithm

- The combination of the decisions done before will always be considered as the origin of a conflict
- Backtracking to the recursion level of the last “branching” in which one case for a variable assignment has not been explored yet
- If such a recursion level does not exist, the given CNF formula is unsatisfiable
From DLL to modern SAT Algorithms

Conflict analysis & backtracking

- Modern SAT algorithms
  - Complex analysis of the conflict setting, because not all “branchings” done before have to be involved in the current conflict
  - Learning of a conflict clause via resolution to avoid running into the same conflict again
  - (Non-)chronological backtracking according to the derived conflict clause
  - If a conflict occurs on decision level 0, the given CNF formula is unsatisfiable
Main techniques of today’s SAT solvers

- Preprocessing
- In turn...
  - Choose the next decision variable
  - Boolean constraint propagation / unit propagation
  - If necessary, conflict analysis & backtracking

- At some fixed points during the search process
  - Unlearning (of some conflict clauses)
  - Restarts
  - Inprocessing

- In case of a satisfiable CNF formula
  - Output the satisfying variable assignment ⇒ model
Modern SAT Algorithms

```c
bool SequentialSatEngine(CNF F)
{
    if (PreprocessCNF(F) == CONFLICT) // Preprocessing the CNF formula
        return UNSATISFIABLE; // Problem unsatisfiable
    while (true)
    {
        if (DecideNextBranch()) // Choice of the next unassigned variable
            // Boolean Constraint Propagation
            // Conflict analysis
            // Cancel the „incorrect“ assignment
            return UNSATISFIABLE; // Problem unsatisfiable
        else
            return SATISFIABLE; // All variables assigned, problem satisfiable
    }
}
```

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Modern SAT Algorithms

```cpp
bool SequentialSatEngine(CNF F)
{
    if (PreprocessCNF(F) == CONFLICT) { return UNSATISFIABLE; }
    while (true) {
        if (DecideNextBranch()) {
            while (BCP() == CONFLICT) {
                BLevel = AnalyzeConflict();
                if (BLevel > 0) {
                    Backtrack(BLevel);
                } else {
                    return UNSATISFIABLE;
                }
            }
        } else {
            return SATISFIABLE;
        }
    }
}
```

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Preprocessing

■ Goal

■ Reduce the formula’s size in terms of clauses and literals to speed up the search process

■ Observation from the experience

■ As a rule of thumb, the size of a formula is related to the time necessary for the SAT algorithm to solve it

■ Identification & preprocessing of unit clauses within the original set of clauses belong to the common operations done in modern SAT algorithms

■ It is very important to find a good compromise between the additional effort required by preprocessing and the expected saving during the search process
Preprocessing

Unit Propagation Lookahead (UPLA)

- Fix a variable $x_i$ to 0, check implications; then change its value to $x_i = 1$, check implications. Simplify the formula exploiting the following consequences:
  - $(x_i = 0 \rightarrow \text{conflict}) \land (x_i = 1 \rightarrow \text{conflict}) \Rightarrow \text{UNSAT}$
  - $(x_i = 0 \rightarrow \text{conflict}) \Rightarrow x_i = 1$
  - $(x_i = 1 \rightarrow \text{conflict}) \Rightarrow x_i = 0$
  - $(x_i = 0 \rightarrow x_j = 1) \land (x_i = 1 \rightarrow x_j = 1) \Rightarrow x_j = 1$
  - $(x_i = 0 \rightarrow x_j = 0) \land (x_i = 1 \rightarrow x_j = 0) \Rightarrow x_j = 0$
  - $(x_i = 0 \rightarrow x_j = 0) \land (x_i = 1 \rightarrow x_j = 1) \Rightarrow x_i \equiv x_j$
Preprocessing

Unit Propagation Lookahead (UPLA)

- **Advantage**
  - Built on top of the components already available in the solver

- **Disadvantages**
  - Requires binary clauses in the original formula
  - Necessary to extend the model when e.g. $x_i \equiv x_j$ is detected and all the occurrences of $x_i$ are substituted with $x_j$
  - In general quite time consuming, in particular if all the variables are tested

\[ (x \lor y) \]
\[ x = 0 \implies y = 1 \]
Preprocessing

Application of resolution

- Advantages
  - No particular kind of clauses necessary in the original formula
  - Usually, simplifies effectively within a manageable time

- Disadvantages
  - In case of a satisfiable CNF formula, model extension required

- Techniques (SatELite)
  - Self-subsuming resolution
  - Elimination by clause distribution
  - Variable elimination by substitution
  - Forward subsumption
  - Backward subsumption
Preprocessing

Self-subsuming resolution

- Original formula
  \[ F = (x_1 \lor \neg x_3) \land (x_1 \lor x_2 \lor x_3) \land \ldots \]

- Resolution applied to the first two clauses
  \[ (x_1 \lor \neg x_3) \otimes_{x_3} (x_1 \lor x_2 \lor x_3) = (x_1 \lor x_2) \]
  \[\Rightarrow (x_1 \lor x_2) \text{ subsumes } (x_1 \lor x_2 \lor x_3) \]
  \[\Rightarrow \text{ Replace } (x_1 \lor x_2 \lor x_3) \text{ with } (x_1 \lor x_2) \]

- Simplified formula
  \[ F' = (x_1 \lor \neg x_3) \land (x_1 \lor x_2) \land \ldots \]

- Saving
  - 1 literal
Preprocessing

Elimination by clause distribution

- Sometimes also called variable elimination
- Original formula:
  \[ F = (x_1 \lor x_2) \land (x_1 \lor \neg x_3) \land (\neg x_1 \lor x_3) \land (\neg x_1 \lor \neg x_2) \]
- Variable elimination applied to \( x_1 \) leads to:
  \[ F' = (x_2 \lor x_3) \land (\neg x_3 \lor \neg x_2) \]
- Saving
  - 1 variable, 2 clauses, 4 literals
- Applied only if it leads to a reduction of the formula’s size
Preprocessing

Variable elimination by substitution

- Original formula
  \[ F = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_4 \lor \neg x_5) \land (\neg x_4 \lor x_5 \lor x_6) \]

- The first three clauses represent an AND gate (\(\leadsto\) Tseitin transformation)
  \[ [(\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2)] \leftrightarrow [x_5 \equiv x_1 \land x_2] \]

- Removing the first three clauses, and replacing the occurrences of \(x_5\) by \(x_1 \land x_2\) in the other clauses leads to
  \[ F' = (x_4 \lor \neg (x_1 \land x_2)) \land (\neg x_4 \lor (x_1 \land x_2) \lor x_6) \]

- Transformation into CNF
  \[ F'' = (x_4 \lor \neg x_1 \lor \neg x_2) \land (\neg x_4 \lor x_1 \lor x_6) \land (\neg x_4 \lor x_2 \lor x_6) \]

- Saving: 1 variable, 2 clauses, 3 literals

- Applied only if it leads to a reduction of the formula’s size

- Procedure for OR, NAND, other “basic gates” quite similar
Preprocessing

**Forward subsumption**
- Test if a clause generated during one of the preprocessing techniques described before is already subsumed by one clause of the current CNF formula

**Backward subsumption**
- Test if a clause generated during one of the preprocessing techniques described before subsumes one (or more) clauses of the current CNF formula

⇒ Remove all the clauses subsumed
Modern SAT Algorithms

```cpp
bool SequentialSatEngine(CNF F)
{
    if (PreprocessCNF(F) == CONFLICT) // Preprocessing the CNF formula
        return UNSATISFIABLE;  
    while (true) // Choice of the next unassigned variable
    {
        if (DecideNextBranch()) // Boolean Constraint Propagation
        {
            while (BCP() == CONFLICT) // Conflict analysis
            {
                BLevel = AnalyzeConflict();
                if (BLevel > 0)
                {
                    Backtrack(BLevel);
                }
                else
                {
                    return UNSATISFIABLE;  
                }
            }
            else
            {
                return SATISFIABLE;  
            }
        }
    }
}
```

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Decision Stack

- Central data structure of modern SAT algorithms
- Decision stack stores the order of the executed assignments
- If a model for a CNF formula could be found, the decision stack stores the satisfying assignment
Each variable assignment has an associated decision level.

- Decision level gets initialized with 0; before a decision is made, it is incremented by one; backtracking decrements the decision level appropriately.

- Decision level 0 plays a special role: It stores only implications from unit clauses in the original formula, but no decisions.

- A conflict on decision level 0 means that the CNF is unsatisfiable.
Decision Stack – First Example

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]
Decision Stack – First Example

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]
Decision Stack – First Example

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]

- Level 0
- Level 1: \(x_1 = 1\)
- Level 2: \(x_2 = 1\)
- Level 3
- Level 4
- Level 5
Decision Stack – First Example

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]

Level 5
Level 4
Level 3
Level 2
Level 1
Level 0

\[x_2 = 1\]
\[x_3 = 0\] Conflict!

\[x_1 = 1\]
Decision Stack – First Example

\[
\neg x_1, \neg x_2, \neg x_3 \land \neg x_1, \neg x_2, x_3 \land \neg x_1, x_2, \neg x_3 \land \neg x_1, x_2, x_3 \land x_1, \neg x_2, \neg x_3
\]
Decision Stack – First Example

\((\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\)
Decision Stack – First Example

\[(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)\]
Decision Stack – First Example

\[ (\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3) \]
Decision Stack – First Example

\[
(\neg x_1, \neg x_2, \neg x_3) \land (\neg x_1, \neg x_2, x_3) \land (\neg x_1, x_2, \neg x_3) \land (\neg x_1, x_2, x_3) \land (x_1, \neg x_2, \neg x_3)
\]
Decision Stack – First Example

⇒ Formula satisfiable with, e.g., $x_1 = 0, x_2 = 0, x_3 = 1$
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

(x₁, x₂) ∧ (x₁, ¬x₃) ∧ (¬x₁, x₃) ∧ (¬x₁, ¬x₂) ∧ (x₃, ¬x₂) ∧ (¬x₃, x₂) ∧ (x₇)
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

\[(x_1, x_2) \land (x_1, \neg x_3) \land (\neg x_1, x_3) \land (\neg x_1, \neg x_2) \land (x_3, \neg x_2) \land (\neg x_3, x_2) \land (x_7)\]
Decision Stack – Second Example

⇒ Formula unsatisfiable due to a conflict on decision level 0
Modern SAT Algorithms

```c
bool SequentialSatEngine(CNF F)
{
    if (PreprocessCNF(F) == CONFLICT) // Preprocessing the CNF formula
        return UNSATISFIABLE; // Problem unsatisfiable
    while (true)
    {
        if (DecideNextBranch()) // Choice of the next unassigned variable
            { // Boolean Constraint Propagation
                while (BCP() == CONFLICT) // Conflict analysis
                { // Cancel the „incorrect“ assignment
                    BLevel = AnalyzeConflict();
                    if (BLevel > 0)
                        { Backtrack(BLevel); }
                    else
                        { return UNSATISFIABLE; }
                }
            }
        else
            return SATISFIABLE; // All variables assigned, problem satisfiable
    }
}
```

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Decision Heuristics

- Have the role of choosing the next decision variable
- Comparable with “case distinction” in the DLL algorithm
- Affects the search process significantly
- Modern SAT algorithms do not test whether the CNF formula is already satisfied during the search, rather it is indirectly guaranteed from assigning all variables without running into a conflict

  - Example: $F = (x_1, x_2, x_3) \land (\neg x_1, x_4)$
  - A satisfying assignment is for example $x_1 = 1, x_4 = 1$
  - Today’s solvers do no test whether $x_1 = x_4 = 1$ already satisfies all the clauses, but assign the remaining variables without generating a conflict (e.g., $x_2 = x_3 = 0$) before they conclude that the CNF is satisfiable
Decision Heuristics

Classical decision heuristics

- Several flavors
  - Dynamic Largest Individual/Combined Sum
  - Maximum Occurrences on Clauses of Minimal Size

- Choice criteria
  - “How often does a still unassigned variable occur in currently unresolved clauses?”
  - Among the unassigned variables, choose the one that occurs most frequently in unresolved clauses
  - In most cases also weighted with the length of those clauses

- These heuristics are quite time consuming, because both the status of each clause and the distribution of the variables within the set of clauses have to be computed and kept up to date

⇒ Computation complexity defined over #clauses
Decision Heuristics

Variable State Independent Decaying Sum (VSIDS)

- Today’s standard method used by almost every SAT solver
- Computation complexity defined over #variables
- No update is mandatory during the backtrack phase
- Each variable $x_i$ has two activity counters $P_{x_i}$ and $N_{x_i}$
- For each literal $L$ in a learned clause $C$ the activity is incremented as follows:

\[
P_{x_i} = P_{x_i} + 1, \text{ if } L = x_i \\
N_{x_i} = N_{x_i} + 1, \text{ if } L = \neg x_i
\]

- The unassigned variable $x_i$ with the highest activity ($P_{x_i}$ or $N_{x_i}$) is chosen as the next decision variable
- Polarity depends on whether $P_{x_i} > N_{x_i}$ holds or not
Variable State Independent Decaying Sum (VSIDS)

- Periodically, the activities are “normalized”, i.e., divided by a constant factor

  ⇒ After the normalization, the recently learned clauses have a higher weight in comparison to the clauses learned before the last normalization process
  ⇒ Takes into account the “history” of the search process

- Several optimizations possible
  - By which amount should the activities be incremented?
  - How often should the normalization take place?
  - By which factor should the activity scores be divided?
bool SequentialSatEngine(CNF F)
{
    if (PreprocessCNF(F) == CONFLICT) // Preprocessing the CNF formula
    {
        return UNSATISFIABLE;  // Problem unsatisfiable
    }

    while (true)
    {
        if (DecideNextBranch()) // Choice of the next unassigned variable
        {
            while (BCP() == CONFLICT) // Boolean Constraint Propagation
            {
                BLevel = AnalyzeConflict(); // Conflict analysis
                if (BLevel > 0)
                {
                    Backtrack(BLevel);  // Cancel the „incorrect“ assignment
                }
                else
                {
                    return UNSATISFIABLE;  // Problem unsatisfiable
                }
            }
        }
        else
        {
            return SATISFIABLE;  // All variables assigned, problem satisfiable
        }
    }
}

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Tasks

- Detect all implications forced by a variable assignment
- Detect conflicts

Comparable to the repeated application of the unit clause rule of the DLL algorithm

Efficient implementation mandatory, because roughly 80% of the runtime of a SAT algorithm is spent by the BCP routine
Boolean Constraint Propagation

General flow

- After every variable assignment, identify the implications that have arisen, and push them into the implication queue
- As long as there are items in the implication queue...
  1. Remove the first element from the queue
  2. Assign to each implied variable its forced truth value
  3. Check which consecutive implications arise, and push them into the implication queue
  4. Check for conflicts
### Boolean Constraint Propagation – Example

**Level 0**
- $x_{23} = 1$
- $x_7 = 1$

**Level 1**
- $x_6 = 0$
- $x_{17} = 0$

**Level 2**
- $x_{13} = 0$
- $x_8 = 1$

**Level 3**
- $x_{19} = 1$
- $x_4 = 1$

**Level 4**
- $x_{54} = 0$

**Level 5**
- $x_{11} = 1$

---

**Implication Queue**

$$F = (x_{23} \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land$$

$$\neg x_{19}, x_4 \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots$$
Boolean Constraint Propagation – Example

Implication Queue

\[
\begin{align*}
&x_{12} = 0 & x_{16} = 1 \\
4 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \}
Boolean Constraint Propagation – Example

Implication Queue

\[ x_{12} = 0 \quad x_{16} = 1 \]

Level 5

\[ x_{11} = 1 \quad x_{12} = 0 \]

Level 4

\[ x_{54} = 0 \]

Level 3

\[ x_{19} = 1 \quad x_{4} = 1 \]

Level 2

\[ x_{13} = 0 \quad x_{8} = 1 \]

Level 1

\[ x_{6} = 0 \quad x_{17} = 0 \]

Level 0

\[ x_{23} = 1 \quad x_{7} = 1 \]

\[ F = (x_{23}) \land (x_{7}, \neg x_{23}) \land (x_{6}, \neg x_{17}) \land (x_{6}, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_{8}) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_{2}) \land (x_{2}, \neg x_{4}, \neg x_{10}) \land \]

\[ (\neg x_{19}, x_{4}) \land (x_{10}, \neg x_{5}) \land (x_{10}, x_{3}) \land (x_{10}, \neg x_{8}, x_{1}) \land (\neg x_{19}, \neg x_{18}, \neg x_{3}) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_{3}, x_{5}) \land \ldots \]

\[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \]
Boolean Constraint Propagation – Example

Implication Queue

\[ x_{12} = 0 \quad x_{16} = 1 \quad x_2 = 0 \]

Level 0

Level 1

Level 2

Level 3

\[ x_{19} = 1 \quad x_4 = 1 \]

Level 4

\[ x_{54} = 0 \]

Level 5

\[ x_{11} = 1 \quad x_{12} = 0 \quad x_{16} = 1 \]

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \]

\[ (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]
Boolean Constraint Propagation – Example

Implication Queue

Level 0

Level 1

Level 2

Level 3

Level 4

Level 5

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \]

\[ \land (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_{18}, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]
Boolean Constraint Propagation – Example

Implication Queue

Level 5
\[ x_{11} = 1 \quad x_{12} = 0 \quad x_{16} = 1 \quad x_{2} = 0 \quad x_{10} = 0 \]

Level 4
\[ x_{5} = 0 \]

Level 3
\[ x_{19} = 1 \quad x_{4} = 1 \]

Level 2
\[ x_{13} = 0 \quad x_{8} = 1 \]

Level 1
\[ x_{6} = 0 \quad x_{17} = 0 \]

Level 0
\[ x_{23} = 1 \quad x_{7} = 1 \]

\[ F = (x_{23}) \land (x_{7}, \neg x_{23}) \land (x_{6}, \neg x_{17}) \land (x_{6}, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_{8}) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_{2}) \land (x_{2}, \neg x_{4}, \neg x_{10}) \land 
\]

\[ (\neg x_{19}, x_{4}) \land (x_{10}, \neg x_{5}) \land (x_{10}, x_{3}) \land (x_{10}, \neg x_{8}, x_{1}) \land (\neg x_{19}, \neg x_{18}, \neg x_{3}) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_{3}, x_{5}) \land \ldots 
\]
Boolean Constraint Propagation – Example

Implication Queue

Level 0

Level 1

Level 2

Level 3

Level 4

Level 5

F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_{2}) \land (x_2, \neg x_{4}, \neg x_{10}) \land \\
(\neg x_{19}, x_4) \land (x_{10}, \neg x_{5}) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_{3}) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_{3}, x_{5}) \land …
Boolean Constraint Propagation – Example

Implication Queue

Level 0

\[ x_{12} = 0 \quad x_{16} = 1 \quad x_2 = 0 \quad x_{10} = 0 \quad x_5 = 0 \quad x_3 = 1 \quad x_1 = 1 \quad x_{18} = 0 \]

Level 5

\[ x_{11} = 1 \quad x_{12} = 0 \quad x_{16} = 1 \quad x_2 = 0 \quad x_{10} = 0 \quad x_5 = 0 \quad x_3 = 1 \]

Level 4

\[ x_{54} = 0 \]

Level 3

\[ x_{19} = 1 \quad x_4 = 1 \]

Level 2

\[ x_{13} = 0 \quad x_8 = 1 \]

Level 1

\[ x_6 = 0 \quad x_{17} = 0 \]

Level 0

\[ x_{23} = 1 \quad x_7 = 1 \]

\[ F = (x_{23}) \land (x_7 \land \neg x_{23}) \land (x_6 \land \neg x_{17}) \land (x_6 \land \neg x_{11} \land \neg x_{12}) \land (x_{13} \land x_8) \land (\neg x_{11} \land x_{13} \land x_{16}) \land (x_{12} \land \neg x_{16} \land \neg x_2) \land (x_2 \land \neg x_4 \land \neg x_{10}) \land \\
(\neg x_{19} \land x_4) \land (x_{10} \land \neg x_5) \land (x_{10} \land x_3) \land (x_{10} \land \neg x_8 \land x_1) \land (\neg x_{19} \land \neg x_{18} \land \neg x_3) \land (x_{17} \land \neg x_1 \land x_{18} \land \neg x_3 \land x_5) \land \ldots \]

1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14
Boolean Constraint Propagation – Example

Implication Queue

\[
\begin{array}{cccccccccc}
& x_{12} = 0 & x_1 = 1 & x_2 = 0 & x_{10} = 0 & x_5 = 0 & x_3 = 1 & x_1 = 1 & x_{18} = 0 & x_{18} = 1
\end{array}
\]

Level 0

\[
\begin{array}{cccccccccc}
& & & & & & & & & \\
& & & & & & & & & \\
\end{array}
\]

Level 1

\[
\begin{array}{cccccccccc}
 x_6 = 0 & & & & & & & & \\
 x_{17} = 0 & & & & & & & & \\
 x_7 = 1 & & & & & & & & \\
\end{array}
\]

Level 2

\[
\begin{array}{cccccccccc}
 x_{13} = 0 & & & & & & & & \\
 x_8 = 1 & & & & & & & & \\
\end{array}
\]

Level 3

\[
\begin{array}{cccccccccc}
 x_{19} = 1 & x_4 = 1 & & & & & & & \\
\end{array}
\]

Level 4

\[
\begin{array}{cccccccccc}
 x_{54} = 0 & & & & & & & & \\
\end{array}
\]

Level 5

\[
\begin{array}{cccccccccc}
 x_{11} = 1 & x_{12} = 0 & x_{16} = 1 & x_2 = 0 & x_{10} = 0 & x_5 = 0 & x_3 = 1 & x_1 = 1 & & \\
\end{array}
\]

\[F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \]

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14
\end{array}
\]

\[
\begin{array}{cccccccccc}
(\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots
\end{array}
\]
Boolean Constraint Propagation – Example

Impactation Queue

\[ x_{12} = 0 \quad x_{16} = 1 \quad x_2 = 0 \quad x_{10} = 0 \quad x_5 = 0 \quad x_3 = 1 \quad x_1 = 1 \quad x_{18} = 0 \quad x_{18} = 1 \]

Level 5

\[ x_{11} = 1 \quad x_{12} = 0 \quad x_{16} = 1 \quad x_2 = 0 \quad x_{10} = 0 \quad x_5 = 0 \quad x_3 = 1 \quad x_1 = 1 \quad x_{18} = 0 \]

Level 4

\[ x_{54} = 0 \]

Level 3

\[ x_{19} = 1 \quad x_4 = 1 \]

Level 2

\[ x_{13} = 0 \quad x_8 = 1 \]

Level 1

\[ x_6 = 0 \quad x_{17} = 0 \]

Level 0

\[ x_{23} = 1 \quad x_7 = 1 \]

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \\
\quad (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_3, x_5) \land \ldots \\
\]
Approaches for the implementation of a BCP routine

- Counter-Based Schemes
- Watched Literals / 2-Literal Watching Scheme
Boolean Constraint Propagation

Counter-Based Schemes

■ 2-Counter Scheme
  ■ Two counters for each clause
    ■ One counter for the literals which satisfy the clause
    ■ One counter for the unassigned literals

■ 1-Counter Scheme
  ■ One counter for each clause to count the number of not falsifying literals

■ Disadvantages
  ■ “Unnecessary” counter updates
  ■ Adjustment of the counter values during backtrack
  ■ Requires a list for each variable and polarity to store all the clauses where the “related literal” (variable having that polarity) occurs
Boolean Constraint Propagation

**Watched Literals**

- For each clause mark two different literals
- **Invariant**
  - Watched literals of a clause are either unassigned or satisfy the clause
- **Advantages in comparison to counter-based schemes**
  - Update operations only when necessary, i.e., when an assignment “breaks” the invariant
  - One list for each variable and polarity (like before), but containing only the clauses currently watched by that literal
- **Disadvantage**
  - Literals of a clause are checked several times
Watched Literals

(a) Initial state

(b) $x_{17} = 0$

c) $x_{5} = 0$

(d) $x_{3} = 1$

(e) $x_{1} = 1 \Rightarrow x_{18} = 1$

(f) $x_{18} = 0 \Rightarrow$ Conflict!
Watched Literals

Possible optimizations

- Always store the watched literals in the first two positions of a clause
  - Allows for a fast access to the “second” watched literal of a clause
  - If the second watched literal satisfies the clause, it is not necessary to find a replacement for the first one (in case the status of the first one switches from unresolved to false)

Nowadays, the BCP procedures of almost all modern SAT solvers are based on watched literals!
bool SequentialSatEngine(CNF F) {
    if (PreprocessCNF(F) == CONFLICT) { // Preprocessing the CNF formula
        return UNSATISFIABLE; } // Problem unsatisfiable
    while (true) {
        if (DecideNextBranch()) { // Choice of the next unassigned variable
            while (BCP() == CONFLICT) { // Boolean Constraint Propagation
                BLevel = AnalyzeConflict(); // Conflict analysis
                if (BLevel > 0) { // Cancel the „incorrect“ assignment
                    Backtrack(BLevel);
                } else { // Problem unsatisfiable
                    return UNSATISFIABLE;
                }
            }
        } else { // All variables assigned, problem satisfiable
            return SATISFIABLE;
        }
    }
}

Not explicitly stated: Inprocessing, unlearning, restarts, model output
Conflict Analysis & Backtracking

DLL algorithm

- The combination of the decisions done before will always be considered as the origin of a conflict.
- Backtracking to the recursion level of the last “branching” in which one case for a variable assignment has not been explored yet (chronological backtracking).
- If such a recursion level does not exist, the given CNF formula is unsatisfiable.
Conflict Analysis & Backtracking

Chronological Backtracking

Level 0: $x_{23} = 1$, $x_7 = 1$
Level 1: $x_6 = 0$, $x_{17} = 0$
Level 2: $x_{13} = 0$, $x_8 = 1$
Level 3: $x_9 = 1$, $x_4 = 1$
Level 4: $x_{54} = 0$
Level 5: $x_{11} = 1$, $x_2 = 0$, $x_{16} = 1$, $x_{10} = 0$, $x_5 = 0$, $x_3 = 1$, $x_1 = 1$, $x_{18} = 0$

$F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{13}) \land (x_{10}, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots$
Conflict Analysis & Backtracking

Modern SAT algorithms

- Complex analysis of the conflict setting, because not all “branchings” done before have to be involved in the current conflict

- Learning of a conflict clause via resolution to avoid running into the same conflict again

- (Non-)chronological backtracking according to the derived conflict clause

- If a conflict occurs on decision level 0, the given CNF formula is unsatisfiable
Conflict Analysis & Backtracking

Implication graph

- Data structure for performing the conflict analysis in today’s SAT solvers
- Directed, acyclic graph
- Nodes represent assignments to variables
- Edges represent which set of assignments have caused an implication
- Implication graph gets updated after every variable assignment and after every backtrack operation
Conflict Analysis & Backtracking

Level 5

\[
\begin{align*}
x_{11} &= 1 \\
x_{12} &= 0 \\
x_{16} &= 1 \\
x_2 &= 0 \\
x_{10} &= 0 \\
x_5 &= 0 \\
x_3 &= 1 \\
x_1 &= 1 \\
x_{18} &= 0 \\
\end{align*}
\]

Level 4

\[
x_{54} = 0
\]

Level 3

\[
\begin{align*}
x_{19} &= 1 \\
x_4 &= 1
\end{align*}
\]

Level 2

\[
\begin{align*}
x_{13} &= 0 \\
x_8 &= 1
\end{align*}
\]

Level 1

\[
\begin{align*}
x_6 &= 0 \\
x_{17} &= 0
\end{align*}
\]

Level 0

\[
\begin{align*}
x_{23} &= 1 \\
x_7 &= 1
\end{align*}
\]

\[
F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \\

(\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land ...
\]

Conflict!
Conflict Analysis & Backtracking

- During the conflict analysis the implication graph gets traversed backwards (in reverse order of the assignments stored by the decision stack) starting from the conflicting point, to allow to compute the succession of resolution steps which finally lead to the conflict clause.

- Different termination criteria for interrupting the resolution steps lead to different conflict clauses.

- Implementations
  - 1UIP (standard technique explained in the following)
  - RelSat
  - Grasp
  - …

First Unique Implication
Conflicts Analysis & Backtracking

\[
F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \\
(\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots
\]
Conflict Analysis & Backtracking

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_7) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \neg x_{19} \land (x_{10}, \neg x_5) \land (x_{10}, \neg x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]

\[ R_1 = (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \otimes x_{18} (\neg x_{19}, \neg x_{18}, \neg x_3) = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \]
Conflict Analysis & Backtracking

\[ F = (x_23) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]

\[ R_1 = (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \otimes_{x_18} (\neg x_{19}, \neg x_{18}, \neg x_3) = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \]

\[ R_2 = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \otimes_{x_1} (x_1, x_{10}, \neg x_8) = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
Conflict Analysis & Backtracking

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]

\[ R_1 = (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \otimes x_{18} (\neg x_{19}, \neg x_{18}, \neg x_3) = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \]

\[ R_2 = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \otimes x_1 (x_1, x_{10}, \neg x_8) = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \]

\[ R_3 = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \otimes x_3 (x_{10}, x_3) = (x_{17}, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
Conflict Analysis & Backtracking

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \\
(\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]

\[ R_1 = (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \otimes x_{18} \ (\neg x_{19}, \neg x_{18}, \neg x_3) = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \]
\[ R_2 = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \otimes x_1 \ (x_1, x_{10}, \neg x_8) = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
\[ R_3 = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \otimes x_3 \ (x_{10}, x_3) = (x_{17}, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
\[ R_4 = (x_{17}, x_5, \neg x_{19}, x_{10}, \neg x_8) \otimes x_5 \ (x_{10}, \neg x_5) = (x_{17}, \neg x_{19}, x_{10}, \neg x_8) \]
Conflict Analysis & Backtracking

\[ F = (x_{23}) \land (x_7, \neg x_{23}) \land (x_6, \neg x_{17}) \land (x_6, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_8) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_2) \land (x_2, \neg x_4, \neg x_{10}) \land \]
\[ (\neg x_{19}, x_4) \land (x_{10}, \neg x_5) \land (x_{10}, x_3) \land (x_{10}, \neg x_8, x_1) \land (\neg x_{19}, \neg x_{18}, \neg x_3) \land (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \land \ldots \]

\[ R_1 = (x_{17}, \neg x_1, x_{18}, \neg x_3, x_5) \otimes x_{18} (\neg x_{19}, \neg x_{18}, \neg x_3) = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \]
\[ R_2 = (x_{17}, \neg x_1, \neg x_3, x_5, \neg x_{19}) \otimes x_1 (x_1, x_{10}, \neg x_8) = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
\[ R_3 = (x_{17}, \neg x_3, x_5, \neg x_{19}, x_{10}, \neg x_8) \otimes x_3 (x_{10}, x_3) = (x_{17}, x_5, \neg x_{19}, x_{10}, \neg x_8) \]
\[ R_4 = (x_{17}, x_5, \neg x_{19}, x_{10}, \neg x_8) \otimes x_5 (x_{10}, \neg x_5) = (x_{17}, \neg x_{19}, x_{10}, \neg x_8) \iff \text{Final conflict clause} \]
Conflict clause: \((x_{17}, \neg x_{19}, x_{10}, \neg x_{8})\)

\[ F = (x_{23} \land (x_{7}, \neg x_{23}) \land (x_{6}, \neg x_{17}) \land (x_{6}, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_{8}) \land (x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_{2}) \land (x_{2}, \neg x_{4}, \neg x_{10}) \land \]

\[ \neg x_{19}, x_{4} \land (x_{10}, \neg x_{5}) \land (x_{10}, x_{3}) \land (x_{10}, \neg x_{8}, x_{1}) \land (\neg x_{19}, x_{18}, \neg x_{3}) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_{3}, x_{5}) \land \ldots \]
Conflict Analysis & Backtracking

Observations

- Conflict analysis according to the 1UIP scheme (First Unique Implication Point) terminates as soon as the computed resolvent contains exactly one literal at the current decision level (the so-called UIP), whereas all other literals were assigned at lower decision levels.

- Conflict clauses represent combinations of variables that will inevitably lead to a conflict.

- Resolution Lemma allows to insert a conflict clause into the CNF formula, and consequently to “prune” the whole search tree by preventing the solver from running into the same conflict again.

- Compared to others, the 1UIP scheme turned out to be the most powerful one (shorter conflict clauses, more effective pruning, faster runtime).
(Non)-chronological backtracking

- In today’s SAT algorithms the backtrack level is determined by the derived conflict clause only.

- The backtrack level matches the maximum decision level among all the literals in the conflict clause except the UIP, which becomes an implication after backtracking.

- Idea: “What would have happened if the conflict clause had already been contained into the original CNF formula?”
Conflict Analysis & Backtracking

(Non-)chronological backtracking

- Procedure
  1. Backtrack down to the given backtrack level
  2. Assign the truth value implied by the UIP (after backtracking, the conflict clause will be automatically a unit clause)
  3. Proceed with the search process

- If a conflict appears at decision level 0, the CNF formula is unsatisfiable
Conflict Analysis & Backtracking

Level 5: $x_{17} = 1$ $x_{19} = 0$ $x_{10} = 0$ $x_{1} = 1$ $x_{18} = 0$

Level 4: $x_{54} = 0$

Level 3: $x_{19} = 1$ $x_{4} = 1$

Level 2: $x_{13} = 0$ $x_{8} = 1$

Level 1: $x_{6} = 0$ $x_{17} = 0$

Level 0: $x_{23} = 1$ $x_{7} = 1$

Conflict clause: $(x_{17}, \neg x_{19}, x_{10}, \neg x_{8})$

Non-Chronological Backtracking

Level 5: $x_{19} = 1$ $x_{4} = 1$ $x_{10} = 1$

Level 4: $x_{13} = 0$ $x_{8} = 1$

Level 3: $x_{6} = 0$ $x_{17} = 0$

Level 2: $x_{23} = 1$ $x_{7} = 1$

Level 0: $x_{23} = 1$ $x_{7} = 1$

Conflict clause: $(x_{17}, \neg x_{19}, x_{10}, \neg x_{8})$

$F = (x_{23}) \land (x_{7}, \neg x_{23}) \land (x_{6}, \neg x_{17}) \land (x_{6}, \neg x_{11}, \neg x_{12}) \land (x_{13}, x_{8}) \land (\neg x_{11}, x_{13}, x_{16}) \land (x_{12}, \neg x_{16}, \neg x_{2}) \land (x_{2}, \neg x_{4}, \neg x_{10}) \land (\neg x_{19}, \neg x_{4}) \land (x_{10}, \neg x_{5}) \land (x_{10}, x_{3}) \land (x_{10}, \neg x_{8}, x_{1}) \land (\neg x_{19}, \neg x_{18}, \neg x_{3}) \land (x_{17}, \neg x_{1}, x_{18}, \neg x_{3}, x_{5}) \land \ldots$
Other Features of modern SAT Solvers

- Unlearning of conflict clauses
- Inprocessing
- Restarts
- Termination guarantees
- Unsatisfiability certificates
- Assumptions
- Incremental SAT solving
- Parallel SAT algorithms
- Incomplete SAT algorithms
Outline

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT
Combinational Equivalence Checking

- **Given**
  - Specification and implementation of a combinatorial circuit

- **Question**
  - Are specification and implementation equivalent?

- **Approach for SAT-based equivalence checking**
  - Generate a so-called Miter from specification and implementation
  - Build a CNF formula from the Miter representation
  - Solve the formula with a SAT algorithm
  - Specification and implementation of a combinatorial circuit are equivalent iff the CNF formula generated from the Miter is unsatisfiable
Miter

⇒ Connect corresponding inputs
Miter

⇒ Link corresponding outputs by EXOR gates
Miter

⇒ Miter circuit
Miter

\[ \Rightarrow M = 1 \iff \text{Specification & implementation not equivalent} \]
Miter

Remarks

- Drafted method can be extended to combinatorial circuits having multiple outputs

- Usually, SAT-algorithms take as input only CNF formulas, that means the Boolean function of the Miter circuit must be translated into a CNF representation $\rightarrow$ Tseitin transformation
Tseitin Transformation

In order to avoid the exponential size of the CNF form obtained from the formula created from the function \( F \) of the circuit, some alternative techniques can be applied:

- Define a **satisfiability equivalent** CNF \( F' \) equivalent to \( F \) that is satisfiable iff \( F \) is satisfiable.
- For each gate output insert an additional variable \( \Rightarrow \) in general the CNF \( F' \) will have variables which do not occur in \( F \).
- For each gate realize a “characteristic function” in CNF which evaluates to 1 for every possible consistent signal configuration.
- Put together the individual gates using an AND connection to obtain the final CNF formula.

\( \Rightarrow \) **Tseitin transformation**
### Tseitin Transformation

<table>
<thead>
<tr>
<th>Gates</th>
<th>Function</th>
<th>CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \rightarrow x_3 \quad x_2$</td>
<td>$x_3 \equiv x_1 \land x_2$</td>
<td>$(\neg x_3 \lor x_1) \land (\neg x_3 \lor x_2) \land (x_3 \lor \neg x_1 \lor \neg x_2)$</td>
</tr>
<tr>
<td>$x_1 \rightarrow x_3 \quad x_2$</td>
<td>$x_3 \equiv x_1 \lor x_2$</td>
<td>$(x_3 \lor \neg x_1) \land (x_3 \lor \neg x_2) \land (\neg x_3 \lor x_1 \lor x_2)$</td>
</tr>
<tr>
<td>$x_1 \rightarrow x_3 \quad x_2$</td>
<td>$x_3 \equiv x_1 \oplus x_2$</td>
<td>$(\neg x_3 \lor x_1 \lor x_2) \land (\neg x_3 \lor \neg x_1 \lor \neg x_2) \land (x_3 \lor \neg x_1 \lor x_2) \land (x_3 \lor x_1 \lor \neg x_2)$</td>
</tr>
<tr>
<td>$x_1 \rightarrow x_2$</td>
<td>$x_2 \equiv \neg x_1$</td>
<td>$(x_2 \lor x_1) \land (\neg x_2 \lor \neg x_1)$</td>
</tr>
</tbody>
</table>
Tseitin Transformation – Example

\[ F_{SK} = (x_1 \land x_2) \lor \neg x_3 \]

\[ F_{SK}^{CNF} = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \]
Tseitin Transformation – Example

\[ F_{SK} = (x_1 \land x_2) \lor \neg x_3 \]

\[ F_{SK}^{CNF} = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \]
Tseitin Transformation – Example

\[ F_{SK} = (x_1 \land x_2) \lor \neg x_3 \]

\[ F_{SK}^{CNF} = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \]
Tseitin Transformation – Example

\[ F_{SK} = (x_1 \land x_2) \lor \neg x_3 \]

\[ F_{SK}^{CNF} = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \]
Important property

- As long as for the CNF representation of each single gate only a constant number of clauses is required, the number of clauses in the final CNF will be linear in the number of gates in the circuit.
Combinational Equivalence Checking – Example

Let the specification and the implementation of a combinatorial circuit be defined as follows:

Question: Are the specification and the implementation equivalent?
Combinational Equivalence Checking – Example

F_M = \neg x_5 \lor x_4 \land \neg x_5 \lor x_2 \land x_5 \lor \neg x_1 \lor \neg x_2 \land x_6 \lor x_3 \land \neg x_6 \lor \neg x_3 \land
\neg x_4 \lor x_5 \land \neg x_4 \lor \neg x_6 \land \neg x_5 \lor x_2 \land \neg x_5 \lor x_3 \land x_7 \lor \neg x_1 \lor \neg x_2 \land x_7 \lor x_8 \land \neg x_7 \lor \neg x_8 \land \neg x_9 \lor x_3 \land \neg x_9 \lor x_8 \land
\neg x_9 \lor \neg x_3 \lor \neg x_8 \land x_9 \lor x_4 \lor \neg x_4 \land \neg M \lor \neg x_4 \lor \neg x_4 \land
\neg M \lor x_4 \lor x_4 \land (M \lor \neg x_4 \lor x_4) \land (M \lor x_4 \lor \neg x_4) \land \neg (M)
Combinational Equivalence Checking – Example

$F_M = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \land (\neg x_7 \lor x_1) \land (\neg x_7 \lor x_2) \land (x_7 \lor \neg x_1 \lor \neg x_2) \land (x_7 \lor x_8) \land (\neg x_7 \lor \neg x_8) \land (\neg x_9 \lor x_3) \land (\neg x_9 \lor x_8) \land (x_9 \lor \neg x_3 \lor \neg x_8) \land (x_9 \lor x'_4) \land (\neg x_9 \lor x'_4) \land (\neg M \lor \neg x_4 \lor \neg x'_4) \land (\neg M \lor x'_4) \land (M \lor x_4 \lor \neg x'_4) \land (M)$

$F_M$ is unsatisfiable $\Rightarrow$ Implementation and specification are equivalent!
Nowadays SAT solvers can handle problems with millions of clauses. But how to compare (large) combinatorial circuits for which SAT methods still fail? ⇒ Structural methods

- Solve several “small” problems instead of one “large” problem
- Various options
  - Compute equivalent gates inside the miter circuit
  - And-Inverter-Graphs (AIGs)
  - …
Observation from real-world instances

- In most cases circuits which have to be compared show structural similarities
  - Example: Only small changes in later design phases
  - In many cases logic optimizations respect hierarchy boundaries
- Thus, changes are not fundamental in most cases
Structural Methods

Observation from real-world instances

- In most cases circuits which have to be compared show structural similarities
  - Example: Only small changes in later design phases
  - In many cases logic optimizations respect hierarchy boundaries
  - Thus, changes are not fundamental in most cases

How can we exploit structural similarities?
Structural Methods

Approach

1. Traverse the circuits which have to be compared from inputs to outputs
   - Identify equivalences at the internal signals of the miter
   - If there are any equivalences, replace equivalent nodes by one (shared) representative

2. Check satisfiability of the simplified miter circuit
Structural Methods – Simple Example

Starting point
Are the internal signals $d$ and $e$ equivalent?
Parts of the miter which are relevant for the proof of $d \equiv e$
Local analysis is sufficient to show that $d \equiv e$
Structural Methods – Simple Example

Simplified miter
Are the internal signals $h$ and $j$ equivalent?
Structural Methods – Simple Example

Parts of the miter which are relevant for the proof of $h \equiv j$
Local analysis is sufficient to show that $h \equiv j$
Structural Methods – Simple Example

More simplified miter
Structural Methods – Simple Example

Does $z = 0$ hold? Are specification and implementation equivalent?
Parts of the miter which are relevant for the proof of $z = 0$
Local analysis is sufficient to show that $z = 0$
⇒ Specification and implementation are equivalent!
Detect potential candidates for pairs of equivalent nodes by simulation with random patterns

- By an (incomplete) simulation of a restricted number of patterns we can only show “non-equivalence”
- Use simulation to partition the nodes into equivalence classes which consist of the nodes with identical simulation results
- Use a complete method (e.g. SAT) to detect equivalent nodes within the computed equivalence classes
Using SAT to prove equivalences

- In order to keep the miter circuit “small”, the inputs of the SAT problem are not necessarily primary inputs, but rather equivalent internal nodes which have already been detected to be equivalent.

- Two nodes are equivalent, if the SAT instance representing the corresponding miter is unsatisfiable.

- If two nodes are proved to be equivalent, then one of the nodes may be replaced by its equivalent counterpart.

- Be careful: If the SAT instance is satisfiable, then this does not necessarily mean that the corresponding nodes are not equivalent!

**False Negatives!**
Structural Methods – Detection of Equivalences

Equivalent nodes can be used as so-called cut points after they have been replaced by a common representative.

- Cut points will be new input variables during miter construction and thus keep the miter “small”.
- If the resulting circuits are equivalent, then the original circuits have already been equivalent.

Problem: Using cut points may lead to so-called “false negatives”, i.e., two equivalent nodes are not classified to be equivalent!
Equivalent nodes can be used as so-called cut points after they have been replaced by a common representative.

- Cut points will be new input variables during miter construction and thus keep the miter “small”
- If the resulting circuits are equivalent, then the original circuits have already been equivalent

Problem: Using cut points may lead to so-called “false negatives”, i.e., two equivalent nodes are not classified to be equivalent!
Structural Methods – Example
Note: Specification and implementation are equivalent
Try to show equivalence of $y_1$ and $y_2$ using cut points
Structural Methods – Example

Assumption: Equivalences $eq_1$, $eq_2$, and $eq_3$ already shown
Structural Methods – Example

Cut the circuits at the internal equivalent signals
Compute the miter depending on “cut variables”
Structural Methods – Example

\[ eq1 \neq 3 \]
\[ d \neq 2 \]
\[ eq2 \]

Corresponding CNF formula satisfiable

\[ \Rightarrow y_1 \text{ and } y_2 \text{ are not equivalent} \]

\[ \Rightarrow \text{Specification and implementation not equivalent} \]

\[ \text{But it is a False Negative!} \]
Structural Methods – Example

Corresponding CNF formula satisfiable

⇒ $y_1$ and $y_2$ not equivalent
⇒ Specification and implementation not equivalent
⇒ But it is a False Negative!
Problem

- New variables at cut points may be assigned to arbitrary values

But…

- The “rightmost” parts of the circuit need only to be equivalent for values at the cut points which can be produced by the “leftmost” parts
Structural Methods – Avoiding False Negatives

- **Do not use cut points**
  - Makes proofs of equivalence for two nodes much more difficult in many cases, since the corresponding SAT problems become significantly “larger”

- **SAT sweeping**
  - In a first step stop at cut points when constructing the miter
  - If necessary (satisfiable CNF) include more parts of the circuit into the SAT problem to check for false negative results
Outline

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT
Automatic Test Pattern Generation

Motivation

- Post-production test is a crucial step
  - Have there been problems during production?
  - Does the circuit contain faults?
- In particular when used in safety-critical applications, every produced chip has to be tested
- Testing comprises more than 40% of costs in semiconductor industry
Automatic Test Pattern Generation

Testing: Experiment on real manufactured chips

- Goal is to check whether the chip behaves correctly
- 1. step: Apply an appropriate test pattern
- 2. step: Analyse the response of the circuit under test
Automatic Test Pattern Generation

- Physical defects are modeled on the Boolean level according to a fault model.

- Fault models are an abstract representation of real defects:
  - Single stuck-at
  - Bridging faults
  - Interconnect opens
  - Path delay faults
  - …

- Automatic Test Pattern Generation (ATPG)
  - Given: Circuit $CUT$ and fault model $FM$
  - Goal: Determine test patterns for (all) faults in $CUT$ wrt. $FM$
Automatic Test Pattern Generation

Single stuck-at fault model (s@)

- \( s@0 \): One line is always at logic 0
- \( s@1 \): One line is always at logic 1
- In total only \((2 \times \text{number_of_signals}_{\text{CUT}})\) faults to be checked
- High amount of real defects detected by the s@ fault model!
Automatic Test Pattern Generation – Typical Flow

Faults:

$f_1$
$f_2$
$f_3$
$f_4$
$f_5$
$f_6$
$f_7$
$f_8$
$f_9$
$f_{10}$
$f_{11}$
$f_{12}$
$f_{13}$
Automatic Test Pattern Generation – Typical Flow

Faults:
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

Patterns:
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

generate random patterns
Automatic Test Pattern Generation – Typical Flow

Faults:
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

Patterns:
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

1. Generate random patterns
2. Simulate generated pattern(s)
Automatic Test Pattern Generation – Typical Flow

**Faults:**
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

**Generate random patterns**

**Patterns:**
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

**Simulate generated pattern(s)**

**All faults classified?**

**End**

**Yes**
Automatic Test Pattern Generation – Typical Flow

Faults:
- \( f_1 \)
- \( f_2 \)
- \( f_3 \)
- \( f_4 \)
- \( f_5 \)
- \( f_6 \)
- \( f_7 \)
- \( f_8 \)
- \( f_9 \)
- \( f_{10} \)
- \( f_{11} \)
- \( f_{12} \)
- \( f_{13} \)

Generate random patterns

Simulate generated pattern(s)

All faults classified?

Choose a fault \( f \) from list

Deterministic TPG for \( f \)

Patterns:
- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
Automatic Test Pattern Generation – Typical Flow

**Usual ATPG-flow**

**Faults:**
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

**Generate random patterns**

**Simulate generated pattern(s)**

**All faults classified?**

**Pattern $p$ found?**

- **Yes:** Mark $f$ as redundant
- **No:** Choose a fault $f$ from list

**Deterministic TPG for $f$**

**Patterns:**
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$
Automatic Test Pattern Generation – Typical Flow

Usual ATPG-flow

Faults:
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

Patterns:
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

1. **generate random patterns**
2. **simulate generated pattern(s)**
3. **all faults classified?**
   - yes: **end**
   - no:
     - **choose a fault $f$ from list**
     - **deterministic TPG for $f$**
     - **mark $f$ as redundant**
     - **pattern $p$ found?**
       - no: **mark $f$ as redundant**
       - yes: **end**
Automatic Test Pattern Generation – Typical Flow

Usual ATPG-flow

Faults:
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

Patterns:
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

1. Choose a fault $f$ from list
2. Determine TPG for $f$
3. Simulate generated pattern(s)
4. Generate random patterns
5. All faults classified?
6. Mark $f$ as redundant
7. Pattern $p$ found?
8. End

VTSA'15 Tobias Schubert – SAT-based Test & Verification
Usual ATPG-flow

Faults:

- \( f_1 \)
- \( f_2 \)
- \( f_3 \)
- \( f_4 \)
- \( f_5 \)
- \( f_6 \)
- \( f_7 \)
- \( f_8 \)
- \( f_9 \)
- \( f_{10} \)
- \( f_{11} \)
- \( f_{12} \)
- \( f_{13} \)

Patterns:

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)
Usual ATPG-flow

Faults:
- \( f_1 \)
- \( f_2 \)
- \( f_3 \)
- \( f_4 \)
- \( f_5 \)
- \( f_6 \)
- \( f_7 \)
- \( f_8 \)
- \( f_9 \)
- \( f_{10} \)
- \( f_{11} \)
- \( f_{12} \)
- \( f_{13} \)

Patterns:
- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

1. generate random patterns
2. simulate generated pattern(s) (fault dropping)
3. mark \( f \) as detected; add \( p \) to test set
4. mark \( f \) as redundant
5. all faults classified?
6. choose a fault \( f \) from list
7. deterministic TPG for \( f \)
8. pattern \( p \) found?
9. yes: end
10. no: go back to choose a fault
11. yes: mark \( f \) as redundant; add \( p \) to test set
12. yes: all faults classified?
13. no: generate random patterns
14. yes: end
Usual ATPG-flow:

1. Generate random patterns for all faults.
2. Simulate generated pattern(s) (fault dropping).
3. Mark fault as detected; add pattern to test set.
4. If pattern found, go back to step 3; otherwise, mark fault as redundant.
5. Determine if all faults classified?
   - If yes, end.
   - If no, choose a fault from list and go to deterministic TPG for that fault.

ATPG basics:

- VTSA'15 Tobias Schubert – SAT-based Test & Verification
Automatic Test Pattern Generation – Typical Flow

Usual ATPG flow:

1. Generate random patterns for all faults.
2. Choose a fault \( f \) from the list.
3. Determine a deterministic TPG for \( f \).
4. Simulate generated patterns (fault dropping).
5. Mark \( f \) as detected; add pattern(s) to the test set.
6. Mark \( f \) as redundant if no pattern is found.
7. Repeat until all faults are classified.
8. End.
Automatic Test Pattern Generation – Typical Flow

Faults:
- \( f_1 \)
- \( f_2 \)
- \( f_3 \)
- \( f_4 \) – highlighted
- \( f_5 \)
- \( f_6 \)
- \( f_7 \)
- \( f_8 \)
- \( f_9 \)
- \( f_{10} \)
- \( f_{11} \) – highlighted
- \( f_{12} \)
- \( f_{13} \)

Patterns:
- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)

**Typical ATPG-flow**

1. **Faults:**
   - Generate random patterns
   - Simulate generated pattern(s) *(fault dropping)*
   - Mark \( f \) as detected; add \( p \) to test set

2. **Patterns:**
   - Choose a fault \( f \) from list
   - Determine TPG for \( f \)
   - Mark \( f \) as aborted
   - Pattern \( p \) found?
     - Yes
     - Mark \( f \) as redundant
     - Timeout
   - All faults classified?
     - Yes
     - End
     - No

**ATPG basics**
Automatic Test Pattern Generation – Typical Flow

Faults:
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$
- $f_8$
- $f_9$
- $f_{10}$
- $f_{11}$
- $f_{12}$
- $f_{13}$

Patterns:
- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$
- $p_6$

**Usual ATPG-flow**

1. **generate random patterns**
2. **simulate generated pattern(s) (fault dropping)**
3. **mark $f$ as detected; add $p$ to test set**
4. **mark $f$ as redundant**
5. **end**
6. **choose a fault $f$ from list**
7. **deterministic TPG for $f$**
8. **timeout**
9. **pattern $p$ found?**
   - yes
   - no
10. **mark $f$ as aborted**
11. **all faults classified?**
    - yes
    - no
Redundant faults: $s@0$ at $x_3$ is redundant

- Justifying the error requires $x_1 = 1$ and $x_2 = 1$
- But propagating the error to output $x_4$ requires $x_1 = 0$
Main concept of automatic test pattern generation

- **Justify** the fault and **find** a propagation path
Main concept of automatic test pattern generation

- Justify the fault and find a propagation path
Main concept of automatic test pattern generation

- Justify the fault and find a propagation path
Main concept of automatic test pattern generation

- **Justify** the fault and **find** a propagation path
Main concept of automatic test pattern generation

- **Justify** the fault and **find** a propagation path
Automatic Test Pattern Generation

Main concept of automatic test pattern generation

- Justify the fault and find a propagation path

```
  1       1/0
    |      |
    |      1/0
    v
  0       1/0
```

110 detects the fault
Automatic Test Pattern Generation

Several ATPG-Approaches

- Structural methods
  - D-algorithm
  - PODEM
  - FAN
- SAT-based methods
SAT-based ATPG

Main flow

- Construct the miter containing the correct and the faulty circuit
- Encode the miter as CNF & solve the SAT problem
- If the SAT formula is satisfiable we have found a test pattern for the particular fault under consideration
- Otherwise, the fault is redundant
SAT-based ATPG – Example

(a) Correct circuit

(b) Faulty circuit, $s@1$-error at $x_5$
SAT-based ATPG – Example
SAT-based ATPG – Example

Conversion to CNF

\( x_5 \)

\( 0 \)

\( x_6 \)

\( 1 \)

\( x_5' \)

\( x_4' \)

\( M \)
SAT-based ATPG – Example

Conversion to CNF

\[ (\neg x_5) \]
SAT-based ATPG – Example
SAT-based ATPG – Example
SAT-based ATPG – Example
$F_M = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \land (x'_4 \lor \neg x'_5) \land (x'_4 \lor \neg x'_6) \land (\neg x'_4 \lor x'_5 \lor x_6) \land (\neg M \lor x_4 \lor x'_4) \land (\neg M \lor \neg x_4 \lor \neg x'_4) \land (M \lor \neg x_4 \lor x'_4) \land (M \lor x_4 \lor \neg x'_4) \land (M) \land (\neg x_5) \land (x'_5)$

"Justify firing $S \oplus 1$ at $x_5"
SAT-based ATPG – Example

\[ F_M = (\neg x_5 \lor x_1) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \land (x'_4 \lor \neg x'_5) \land (x'_4 \lor \neg x_6) \land (\neg x'_4 \lor x'_5 \lor x_6) \land (\neg M \lor x_4 \lor x'_4) \land (\neg M \lor \neg x_4 \lor x'_4) \land (M \lor x_4 \lor \neg x'_4) \land (M) \land (\neg x_5) \land (x'_5) \]

\[ F'_M = (\neg x_1 \lor \neg x_2) \land (x_3) \land (\neg x_6) \land (x'_4) \land (\neg x_4) \land (M) \land (\neg x_5) \land (x'_5) \]
SAT-based ATPG – Example

\[ F_M = (\neg x_1 \lor x_2) \land (\neg x_5 \lor x_2) \land (x_5 \lor \neg x_1 \lor \neg x_2) \land (x_6 \lor x_3) \land (\neg x_6 \lor \neg x_3) \land (x_4 \lor \neg x_5) \land (x_4 \lor \neg x_6) \land (\neg x_4 \lor x_5 \lor x_6) \land (x'_4 \lor \neg x'_5) \land (x'_4 \lor \neg x_6) \land (\neg x'_4 \lor x'_5 \lor x_6) \land (\neg M \lor x_4 \lor x'_4) \land (\neg M \lor \neg x_4 \lor x'_4) \land (M \lor \neg x_4 \lor x'_4) \land (M \lor x_4 \lor \neg x'_4) \land (M) \land (\neg x_5) \land (x'_5) \]

\[ F'_M = (\neg x_1 \lor \neg x_2) \land (x_3) \land (\neg x_6) \land (x'_4) \land (\neg x_4) \land (M) \land (\neg x_5) \land (x'_5) \]

Test set: \((x_1, x_2, x_3) = \{(0, 0, 1), (1, 0, 1), (0, 1, 1)\}\)
SAT-based ATPG – Adding Structural Information

Additional Logic

$X_7\, X_1\, X_2\, X_3\, X_8\, X_5\, X_6\, X_4\, X_9\, X_{10}\, X_{11}$
SAT-based ATPG – Adding Structural Information

Additional Logic

\[ x_3 \times x_5 \times x_6 \times x_7 \times x_8 \times x_9 \times x_{10} \times x_{11} \]

s@1-error
SAT-based ATPG – Adding Structural Information

Additional Logic

$\neg x_1 \land x_2 \land x_3 \land x_5 \land x_6 = 0/1$

$\neg x_7 
\neg x_2 \land 0 = 0$

$\neg x_3 \land 1 = 0$

$S@1$-error

$0/1$

$0/1$

$0$

$0$

$0$
SAT-based ATPG – Adding Structural Information

Adding structural information to the CNF

Additional Logic

\[ x_1 \]
\[ x_2 \]
\[ x_3 \]
\[ x_4 \]
\[ x_5 \]
\[ x_6 \]
\[ x_7 \]
\[ x_8 \]
\[ x_9 \]
\[ x_{10} \]
\[ x_{11} \]

s@1-error
SAT-based ATPG – Adding Structural Information

Add \((x_7, x_8)\) to the CNF
SAT-based ATPG – Adding Structural Information

Add \((x_7, x_8)\) to the CNF
SAT-based ATPG – Cone-of-Influence Reduction

Circuit under Test

s@−error

×
Which inputs might be relevant for justifying the fault?
SAT-based ATPG – Cone-of-Influence Reduction

Which outputs might be on the propagation path?
What about side-effects?
⇒ Only the “brown” parts have to be transformed into CNF!
SAT-based ATPG – Testing of Sequential Circuits

Inputs

Combinational Circuit

Outputs (Mealy machine)

Present state

State

FF

Next state

Combinational Circuit

Outputs (Moore machine)
SAT-based ATPG – Testing of Sequential Circuits

Problems specific wrt. test of sequential circuits

- Initialization
  - Circuit’s state at the beginning of test application might be unknown

- Counters
  - Setting a counter to a specific value might take a lot of clock cycles

- Complexity of test generation
  - Finding a sequence to distinguish between a faulty and a fault-free chip might require a large number of state transitions
SAT-based ATPG – Testing of Sequential Circuits

Problems specific wrt. test of sequential circuits

- Initialization
  - Circuit’s state at the beginning of test application might be unknown

- Counters
  - Setting a counter to a specific value might take a lot of clock cycles

- Complexity of test generation
  - Finding a sequence to distinguish between a faulty and a fault-free chip might require a large number of state transitions

⇒ Practical methods reduce sequential to combinatorial ATPG
⇒ Solution: “Design for Testability”-techniques within the chips
⇒ Example: Scan-based designs
SAT-based ATPG – Scan-based Designs

- **Scan:** ScanEnable = 1
- **Capture:** ScanEnable = 0

Diagram showing a scan flip-flop (SFF) connected to combinational logic.
SAT-based ATPG – Scan-based Designs

Test flow

1. Scan in data into SFFs
2. Apply test vector to PIs
3. Perform the test
4. Check POs
5. Scan out & check the data available at SFFs
Sequential Equivalence Checking

Combinational Part

FF_0
FF_1
...
FF_k

Outputs (Mealy Machine)

Current State
Next State

Specification
Implementation

Inputs

Combinational Part

FF_0
FF_1
...
FF_k

Outputs (Mealy Machine)

Current State
Next State

Implementation

VTSA'15 Tobias Schubert – SAT-based Test & Verification 114 / 192
Sequential Equivalence Checking

Combinational Part

FF_0
FF_1
... 
FF_k

Outputs (Mealy Machine)

Implementations

Specification

Inputs

Current State

Next State
Sequential Equivalence Checking

Input: No
What can we do with equivalence checking of sequential circuits?

- Functional equivalence of two sequential circuits (in general) provable
- We cannot prove with equivalence checking whether a circuit satisfies a more abstract specification, which is not given as a sequential circuit or a deterministic finite automaton!

Examples for such abstract specifications are

- Safety properties
- Liveness properties

⇒ New specification language(s) for timed properties and in connection with that new proof methods are necessary!
Preliminaries – Kripke Structure

To model computational runs of a sequential circuit, Kripke structures (also referred to as temporal structures) can be used:

**Definition (Kripke structure, temporal structure)**

A Kripke structure $M$ is a 4-tuple $M := (S, I, R, L)$ consisting of:

- a finite set $S$ of states
- a set $\emptyset \neq I \subseteq S$ of initial states
- a transition relation $R \subseteq S \times S$
  with $\forall s \in S \exists t \in S : (s, t) \in R$, and
- a labeling function $L : S \rightarrow 2^V$, where $V$ is a set of propositional variables (atomic formulas, atomic propositions).

**Atomic propositions** are observable elementary properties of states, like “a timeout has occurred”, “a request has been made”

Using such a temporal structure, we can derive all possible computational runs. They are obtained by “unrolling” the Kripke structure according to its transition relation $R$.
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
- \( G \phi \): Formula \( \phi \) holds in every state on the path ("globally" or "always")
- \( F \phi \): Formula \( \phi \) holds in some state on the path ("finally" or "eventually")
- \( X \phi \): Formula \( \phi \) holds in the second state on the path ("next")
- \( \phi U \psi \): Formula \( \phi \) holds in every state on the path until a state is reached where \( \psi \) holds ("until")

Path quantifiers
- \( A \phi \): Formula \( \phi \) holds on all paths starting in this state ("for all paths")
- \( E \phi \): Formula \( \phi \) holds on some path starting in this state ("there exists a path")
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
They make statements about a **single path** of the computation tree:

- \( G \phi \): Formula \( \phi \) holds in every state on the path ("globally" or "always")
- \( F \phi \): Formula \( \phi \) holds in some state on the path ("finally" or "eventually")
- \( X \phi \): Formula \( \phi \) holds in the second state on the path ("next")
- \( \phi U \psi \): Formula \( \phi \) holds in every state on the path until a state is reached where \( \psi \) holds ("until")

Path quantifiers
They make statements about **properties of states**:

- \( A \phi \): Formula \( \phi \) holds on all paths starting in this state ("for all paths")
- \( E \phi \): Formula \( \phi \) holds on some path starting in this state ("there exists a path")
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
They make statements about a single path of the computation tree:

- $\mathbf{G}\varphi$: Formula $\varphi$ holds in every state on the path ("globally" or "always")

Path quantifiers
They make statements about properties of states:
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators

They make statements about a single path of the computation tree:

- $\mathbf{G}\varphi$: Formula $\varphi$ holds in every state on the path ("globally" or "always")
- $\mathbf{F}\varphi$: Formula $\varphi$ holds in some state on the path ("finally" or "eventually")

Path quantifiers

They make statements about properties of states:
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
They make statements about a single path of the computation tree:

- $G\varphi$: Formula $\varphi$ holds in every state on the path (“globally” or “always”)
- $F\varphi$: Formula $\varphi$ holds in some state on the path (“finally” or “eventually”)
- $X\varphi$: Formula $\varphi$ holds in the second state on the path (“next”)

Path quantifiers
They make statements about properties of states:
Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
They make statements about a single path of the computation tree:

- \( G\varphi \): Formula \( \varphi \) holds in every state on the path ("globally" or "always")
- \( F\varphi \): Formula \( \varphi \) holds in some state on the path ("finally" or "eventually")
- \( X\varphi \): Formula \( \varphi \) holds in the second state on the path ("next")
- \( \varphi U\psi \): Formula \( \varphi \) holds in every state on the path until a state is reached where \( \psi \) holds ("until")

Path quantifiers
They make statements about properties of states:
Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators

They make statements about a single path of the computation tree:

- \( G \varphi \): Formula \( \varphi \) holds in every state on the path ("globally" or "always")
- \( F \varphi \): Formula \( \varphi \) holds in some state on the path ("finally" or "eventually")
- \( X \varphi \): Formula \( \varphi \) holds in the second state on the path ("next")
- \( \varphi U \psi \): Formula \( \varphi \) holds in every state on the path until a state is reached where \( \psi \) holds ("until")

Path quantifiers

They make statements about properties of states:

- \( A \varphi \): Formula \( \varphi \) holds on all paths starting in this state ("for all paths")
Temporal propositional logic = Propositional logic + Temporal operators

Linear temporal operators
They make statements about a single path of the computation tree:
- \( G\varphi \): Formula \( \varphi \) holds in every state on the path ("globally" or "always")
- \( F\varphi \): Formula \( \varphi \) holds in some state on the path ("finally" or "eventually")
- \( X\varphi \): Formula \( \varphi \) holds in the second state on the path ("next")
- \( \varphi U\psi \): Formula \( \varphi \) holds in every state on the path until a state is reached where \( \psi \) holds ("until")

Path quantifiers
They make statements about properties of states:
- \( A\varphi \): Formula \( \varphi \) holds on all paths starting in this state ("for all paths")
- \( E\varphi \): Formula \( \varphi \) holds on some path starting in this state ("there exists a path")
Property/Model Checking in a Nutshell

Model $M$
(Kripke Structure)

Property $\varphi$
(Temporal Logic)

Model Checker

$M \models \varphi$!

Counterexample
Property/Model Checking in a Nutshell

Model $M$

$s_0 \models E(pUq)!$

$\varphi = E(pUq)$

Model Checker
SAT-based Bounded Model Checking

Idea

Formulate the existence of paths with certain properties as satisfiability problem

- Only properties which require the existence of paths
  - Certificate or counterexample depending on context
  - E.g.: Counterexamples for safety and liveness
- In general, arbitrarily long paths necessary, but this is not possible in SAT!
- Restriction to finite path lengths ⇒ bounded model checking
Model Checking vs. Bounded Model Checking

Given

- Kripke structure $M$
- Temporal formula $\varphi$ “suited for BMC”
- Maximum unrolling depth $k$

Model Checking

- $M \models \varphi$?

Bounded Model Checking

- $M \models^k \varphi$?
- $\models^k$ means in this context that from the initial states in $M$, the outgoing paths are considered only up to a maximum length $k$
Illustration 2-Bit Counter: Time Frame Expansion
Let $\phi$ be a temporal formula and $k = 1$. $M \models_1 \phi$?
Let $\varphi$ be a temporal Formula and $k = 2$. $M \models_2 \varphi$?
Let $\varphi$ be a temporal Formula and $k = 3$. $M \models_3 \varphi$?
SAT-based Bounded Model Checking

General flow

1. Generate a propositional logic formula from the given Kripke structure $M$, property $\varphi$, and unrolling depth $k$, which is satisfiable iff $M \models_k \varphi$

2. Translate the formula generated above into CNF

3. Solve it with a SAT solver
   - CNF satisfiable $\Rightarrow M \models_k \varphi \Rightarrow$ certificate/counterexample
   - CNF unsatisfiable $\Rightarrow M \not\models_k \varphi \Rightarrow$ no statement can be made regarding $M \models \varphi$

Repeat the steps from 1 to 3 with increasing values for $k$ until either a counterexample is found, or a fixed stopping criterion is met.
Construction of the propositional logic formula

**Definition**

Let $M = (S, I, R, L)$ be a Kripke structure, $\varphi$ a property, and $k$ an unfolding depth. Then the characteristic function $[M, \varphi]_k$ corresponding to $M$, $\varphi$, and $k$ is defined as

$$I(s_0) \land \left[ \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \right] \land \left[ \bigwedge_{s_j \in S} (s_j \rightarrow L(s_j)) \right] \land P_k(\varphi)$$

with

- $I(s_0)$: characteristic fct. of the initial states,
- $R(s_i, s_{i+1})$: characteristic fct. of the transition relation,
- $L(s_j)$: characteristic fct. of the label function $L$,
- $P_k(\varphi)$: characteristic fct. of $\varphi$ at depth $k$. 
Safety

- Specify invariants of the system:
  \[ \text{AG safe} \]

- BMC-formulation for refuting safety (= proving \( \text{EF} \neg \text{safe} \)):
  \[
  l(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i,s_{i+1}) \land \neg \text{safe}(s_k)
  \]
Types of Properties – Liveness

Liveness

- Specified in temporal logic:
  \[ \text{AF} \text{good} \]

- Refutation of liveness (= proving \( \text{EG} \neg \text{good} \)) requires infinitely long paths!

- If \( \text{AF} \text{good} \) is violated, there is a “lasso” on which all states satisfy \( \neg \text{good} \)

- BMC-formulation:

  \[ I(s_0) \land \bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \land \bigwedge_{i=0}^{k} \neg \text{good}(s_i) \land \bigvee_{l=0}^{k} (s_l = s_{k+1}) \]

VTSA'15  Tobias Schubert – SAT-based Test & Verification 125 / 192
Requirement: State (1, 1) may not reached, or later an overflow will occur, i.e. the following must hold:

\[ \text{AG}(\neg(b \land a)) \iff \neg\text{EF}(b \land a) \]
Requirement: State \((1, 1)\) may not reached, or later an overflow will occur, i.e. the following must hold:

\[
\text{AG}(\neg (b \land a)) \iff \neg \text{EF}(b \land a)
\]

Possible query: Can one reach \((1, 1)\) from the initial state \((0, 0)\) in \(\leq 2\) steps?
BMC Example Safety – 2-Bit Counter

**Requirement:** State \((1,1)\) may not reached, or later an overflow will occur, i.e. the following must hold:

\[
AG(\neg(b \land a)) \iff \neg EF(b \land a)
\]

**Possible query:** Can one reach \((1,1)\) from the initial state \((0,0)\) in \(\leq 2\) steps?

\[
\Rightarrow M \models_2 \varphi \text{ with } \varphi = EF(b \land a)\
\Rightarrow I(s_0) = \neg b_0 \land \neg a_0\
\Rightarrow R(s_0, s_1) = (b_1 \iff (b_0 \oplus a_0)) \land (a_1 \iff \neg a_0)\
\Rightarrow R(s_1, s_2) = (b_2 \iff (b_1 \oplus a_1)) \land (a_2 \iff \neg a_1)\
\Rightarrow P_2(\varphi) = (b_0 \land a_0) \lor (b_1 \land a_1) \lor (b_2 \land a_2)\
\Rightarrow \lbrack M, \varphi \rbrack_2 = I(s_0) \land R(s_0, s_1) \land R(s_1, s_2) \land P_2(\varphi)\
\Rightarrow \lbrack M, \varphi \rbrack_2 = 0\
\Rightarrow \text{Starting from } (0,0), (1,1) \text{ cannot reached in max. 2 steps } \Rightarrow M \not\models_2 \varphi!
Requirement: State \((1, 1)\) may not reached, or later an overflow will occur, i.e. the following must hold:

\[
\text{AG}(\neg(b \land a)) \iff \neg\text{EF}(b \land a)
\]

Possible query: Can one reach \((1, 1)\) from the initial state \((0, 0)\) in \(\leq 2\) steps?

\[
\Rightarrow M \models_2 \varphi \text{ with } \varphi = \text{EF}(b \land a)\?
\]

\[
\Rightarrow I(s_0) = \neg b_0 \land \neg a_0
\]

\[
\Rightarrow R(s_0, s_1) = (b_1 \leftrightarrow (b_0 \oplus a_0)) \land (a_1 \leftrightarrow \neg a_0)
\]

\[
\Rightarrow R(s_1, s_2) = (b_2 \leftrightarrow (b_1 \oplus a_1)) \land (a_2 \leftrightarrow \neg a_1)
\]

\[
\Rightarrow P_2(\varphi) = (b_0 \land a_0) \lor (b_1 \land a_1) \lor (b_2 \land a_2)
\]

\[
\Rightarrow [M, \varphi]_2 = I(s_0) \land R(s_0, s_1) \land R(s_1, s_2) \land P_2(\varphi)
\]

\[
\Rightarrow [M, \varphi]_2 = 0
\]

\[
\Rightarrow \text{Starting from } (0, 0), (1, 1) \text{ cannot reached in max. } 2 \text{ steps } \Rightarrow M \not\models 2 \varphi!
\]

But: \(M \not\models \text{AG}(\neg(b \land a)) \iff M \not\models \neg\text{EF}(b \land a)\)!
Requirement: State \((1, 1)\) must be reachable from every state, i.e. the following must hold:

\[ AF(b \land a) \Leftrightarrow \neg EG(\neg(b \land a)) \]
**BMC Example Liveness – Modified 2-Bit counter**

**Requirement:** State \((1, 1)\) must be reachable from every state, i.e. the following must hold:

\[
\text{AF}(b \land a) \Leftrightarrow \neg \text{EG}((b \land a))
\]

**Counterexample** exists iff from the initial state \((0, 0)\) there exists a path of length \(k\) that belongs to a cycle, and in no state of this path \((b \land a)\) holds. Given \(k = 2\) and \(\varphi = \text{EG}((b \land a))\):

\[
\begin{align*}
&\text{I}(s_0) = \neg b_0 \land \neg a_0 \\
&\text{R}(s_i, s_{i+1}) = ((b_i+1 \leftrightarrow b_i) \land (a_i+1 \leftrightarrow \neg a_i)) \lor (b_i+1 \land \neg a_i+1 \land b_i \land \neg a_i)
\end{align*}
\]

\[
\begin{align*}
&\text{P}_2(\varphi) = \neg b_0 \lor \neg a_0 \land \neg b_1 \lor \neg a_1 \land \neg b_2 \lor \neg a_2
\end{align*}
\]

\[
\begin{align*}
&\text{J}_M, \varphi_K^2 = \text{I}(s_0) \land \left(\bigwedge_{i=0}^{2} \text{R}(s_i, s_{i+1}) \lor \bigvee_{i=0}^{2} [s_3 \equiv s_i] \land \text{P}_2(\varphi)\right)
\end{align*}
\]

\[
\begin{align*}
&\text{counterexample found!}
\end{align*}
\]
BMC Example Liveness – Modified 2-Bit counter

**Requirement:** State \((1, 1)\) must be reachable from every state, i.e. the following must hold:

\[
AF(b \land a) \iff \neg EG(\neg (b \land a))
\]

**Counterexample** exists iff from the initial state \((0, 0)\) there exists a path of length \(k\) that belongs to a cycle, and in no state of this path \((b \land a)\) holds. Given \(k = 2\) and \(\varphi = EG(\neg (b \land a))\):

\[
I(s_0) = \neg b_0 \land \neg a_0
\]

\[
R(s_i, s_{i+1}) = ((b_{i+1} \leftrightarrow (b_i \oplus a_i)) \land (a_{i+1} \leftrightarrow \neg a_i)) \lor (b_{i+1} \land \neg a_{i+1} \land b_i \land \neg a_i) \text{ with } i = 0, 1, 2
\]

\[
P_2(\varphi) = (\neg b_0 \lor \neg a_0) \land (\neg b_1 \lor \neg a_1) \land (\neg b_2 \lor \neg a_2)
\]

\[
[s_3 \equiv s_i] = (b_3 \leftrightarrow b_i) \land (a_3 \leftrightarrow a_i) \text{ with } i = 0, 1, 2
\]

\[
\llbracket M, \varphi \rrbracket_2 = I(s_0) \land \left[ \bigwedge_{i=0}^{2} R(s_i, s_{i+1}) \right] \land \left[ \bigvee_{i=0}^{2} [s_3 \equiv s_i] \right] \land P_2(\varphi)
\]

\[
\llbracket M, \varphi \rrbracket_2 = \neg b_0 \land \neg a_0 \land \neg b_1 \land a_1 \land b_2 \land \neg a_2 \land b_3 \land \neg a_3
\]

⇒ Counterexample found!
SAT-based Bounded Model Checking

- BMC can be used to disprove invariants $\text{AG} \phi$
  - ... by proving $\text{EF} \neg \phi$ considering paths of length $k$
  - If paths longer than $k$ are needed for the proof, then BMC fails

- BMC can be used to disprove liveness properties like $\text{AF} \phi$
  - ... by proving $\text{EG} \neg \phi$ considering “lassos” of length $k$
  - If lassos longer than $k$ are needed for the proof, then BMC fails

- In the following we restrict ourselves to invariants / safety properties
Usage of BMC to falsify Safety Properties

**Idea:** Restrict system behavior to runs of some given bounded length, i.e. runs with a bounded number of transition steps
Usage of BMC to falsify Safety Properties

Idea: If the restricted system is unsafe (i.e. violates some safety property, state invariant) then the original system is unsafe, too.
Usage of BMC in the Verification Domain

- Initial state $I$, transition relation $T$, property $P$
- Iterative unrolling of the system for $k = 0, 1, \ldots, K$ up to a given maximal unrolling depth $K$

\[
BMC_k = I^0 \land \bigwedge_{i=0}^{k-1} T^{i,i+1} \land \neg P^k
\]

- Convert $BMC_k$ into CNF by Tseitin transformation and solve it using a SAT solver
  - CNF satisfiable $\Rightarrow$ Invariant condition $P$ violated after $k$ steps
  - CNF unsatisfiable $\Rightarrow$ no conclusion, next iteration step
Some Remarks

- Typically, BMC is used as an efficient means to find errors in a system $M$, i.e. is there a $k > 0$ such that we can reach a state violating $\varphi$ for a given invariant $\textbf{AG}\varphi$?

- BMC is really efficient if there is a short error path

- Without extensions it is not possible to prove that $\varphi$ holds for all reachable states

- Bounded Model Checking $\rightarrow$ Model Checking
  - Computing the “radius” of the Kripke structure
  - $k$-induction
  - Craig interpolation
The main part of the formula remains unchanged

- $\neg P^i$ has to be removed
- $T^{i,i+1} \land \neg P^{i+1}$ has to be added

How to profit from the similarity between those problems?
Incremental SAT Solving

- In many practical applications – not only in the area of BMC – often several SAT instances are generated to solve a real-world problem.
- Generated SAT instances are often very similar and contain identical subformulas.
- Idea: Instead of constructing and solving each instance separately, the SAT formula is processed incrementally.
- Knowledge learnt so far (conflict clauses, variable activity, ...) can be re-used in later instances.
- Standard feature of all modern SAT solvers.
Incremental SAT Solving

Main idea

- Make use of the knowledge learnt in the previous instance by re-using the learnt conflict clauses

Question

- Is this always allowed?
Incremental SAT Solving

- **Idea:** Make use of the knowledge learnt in the previous instance by re-using the learnt conflict clauses.

- **Question:** Is this always allowed?

- **Observation**
  - If $c$ is a conflict clause for SAT instance $A$ with CNF $CNF_A$, then $CNF_A \Rightarrow c$
  - If instance $B$ results from $A$ just by adding clauses (i.e. $CNF_B \supseteq CNF_A$), then $CNF_B \Rightarrow c$ holds as well
  - Conflict clauses be may re-used then
  - But what if $CNF_B \supseteq CNF_A$ does not hold?
Incremental SAT Solving

- **General case:** $CNF_A$ contains clauses that do not occur in $CNF_B$ anymore

- Now we need for each conflict clause $c$ the information about the set of original clauses it was derived from

- **Remember:** Conflict clauses result from original and/or conflict clauses by resolution ($\Rightarrow$ implication graph)

  $\Rightarrow$ Conflict clauses which are derived from original clauses in $CNF_A \setminus CNF_B$ are not allowed to be added to $CNF_B$!
Illustration: Re-using Clauses
Illustration: Re-using Clauses

Original Clauses
Learned Clauses
Illustration: Re-using Clauses

Original Clauses
Learned Clauses
Incremental SAT Solving with Assumptions

In general, storing which conflict clause depends on which original clauses is too expensive! Here is the most common approach to solve the problem:

**Activation variables and assumptions**

- Use “special” new de-activation variables $d_i$
- For clauses $c$ which should be removable from the clause set, a positive de-activation literal is added: $c := c \cup d_i$
- There are only positive occurrences of de-activation variables!
- Turning $c$ on and off:
  - Turning on by $d_i = 0$
  - Turning off by $d_i = 1$

Example

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

Initial formula $\phi_0 / \neg d_0 = (a \lor b) \land (\neg c \lor d) \land (b \lor d_0)$

Incr. step 0 $\phi_1 / d_0, \neg d_1 = (a \lor b) \land (\neg c \lor d) \land (b \lor d_0) \land (d \lor d_1)$
Incremental SAT Solving with Assumptions

In general, storing which conflict clause depends on which original clauses is too expensive! Here is the most common approach to solve the problem:

**Activation variables and assumptions**

- Use “special” new de-activation variables \( d_i \)
- For clauses \( c \) which should be removable from the clause set, a positive de-activation literal is added: \( c := c \cup d_i \)
- There are only positive occurrences of de-activation variables!
- Turning \( c \) on and off:
  - Turning on by \( d_i = 0 \)
  - Turning off by \( d_i = 1 \)

**Example**

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

Initial formula

\[
\varphi_0 / \neg d_0 = (a \lor b) \land (\neg c \lor d) \land (b \lor d_0)
\]

incr. step 0

\[
\varphi_1 / d_0, \neg d_1 = (a \lor b) \land (\neg c \lor d) \land (b \lor d_0) \land (d \lor d_1)
\]

incr. step 1
Activation variables and assumptions

- ...  
- De-activation variables are assigned by assumptions before SAT solving (activating / de-activating clauses)  
- Assumptions can not be changed during SAT solving (Note: Unit clauses and assumptions are not the same!)

Important observation: All conflict clauses resulting from $c \cup d_i$ by resolution contain literal $d_i$

$\Rightarrow$ If $c \cup d_i$ is turned off in the next run, i.e., $d_i$ is set to 1 by assumption, then all conflict clauses depending on $c \cup d_i$ are turned off as well!
Incremental SAT Solving and BMC

\[ I^0 \land T^{0,1} \land T^{1,2} \land \ldots \land T^{k-1,k} \land \neg P^k \]

\[ k = i : \quad I^0 \land T^{0,1} \land T^{1,2} \land \ldots \land T^{i-1,i} \land \neg P^i \]
\[ k = i + 1 : \quad I^0 \land T^{0,1} \land T^{1,2} \land \ldots \land T^{i-1,i} \land T^{i,i+1} \land \neg P^{i+1} \]

- Add de-activation literal \( d_i \) for each clause representing \( \neg P^i \)
- For \( k = i \) activate \( \neg P^i \) by assumption \( d_i = 0 \)
- For \( k > i \) de-activate \( \neg P^i \) by assumption \( d_i = 1 \)
- All knowledge / conflict clauses learnt for \( k = i \) can be re-used (except the knowledge depending on \( \neg P^i \))
Outline

Applications
- Bounded Model / Property Checking
- Test Pattern Relaxation
- Black Box Verification
- Path Compaction
- Automatic Test Pattern Generation
- Combinational Equivalence Checking
- Security Issues
- Hybrid System Verification
- The End

Core Algorithms
- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT

Combinational Equivalence Checking
Hybrid System Verification
The End
Hybrid Systems

- Typically, embedded systems are characterized by the combination of discrete and continuous variables

**iSAT**

- Satisfiability and BMC checker for quantifier-free Boolean combinations of arithmetic constraints over the reals and integers

\[
\neg b \lor \neg c \\
\land (b \rightarrow \sin(x) \cdot y < 7.2) \\
\land (\sqrt{2x - y} = 8 \lor c) \\
\land (i^2 = 3j - 5)
\]
Satisfiability Modulo Theory – iSAT

iSAT

- Not a “pure” SAT-Modulo-Theory solver

- Can be seen as a generalization of a SAT solver
  - Branch-and-deduce framework inherited from SAT
  - Deduction rule for clauses
    - Unit propagation
  - Deduction rules for arithmetic operators
    - Interval constraint propagation
Satisfiability Modulo Theory – ICP

Interval Constraint Propagation (ICP)

\[ h_1 = z^2, \; z \in [3, 7], \; h_1 \in [-2, 25] \]

\[ z \in [3, 7] \Rightarrow h_1 \geq 9 \Rightarrow h_1 \in [9, 25] \]

\[ h_1 \in [9, 25] \Rightarrow z \leq 5 \Rightarrow z \in [3, 5] \]
There’s no sequence of input values such that $3.14 \leq x \leq 3.15$.

Safety property:

DECL

boole b;
float [0.0, 1000.0] x;

INIT

- Initial state.
x = 2.0;

TRANS

- Transition relation.
b -> x' = x^2 + 1;
!b -> x' = nrt(x, 3);

TARGET

- State(s) to be reached.
x >= 3.14 and x <= 3.15;
Satisfiability Modulo Theory – iSAT

iSAT

- All acceleration techniques known from modern SAT solvers also apply to arithmetic constraints
  - Conflict-driven learning
  - Non-chronological backtracking
  - 2-watched-literal scheme
  - Restarts
  - Conflict clause deletion
  - Efficient decision heuristics
Satisfiability Modulo Theory – iSAT

- Use Tseitin-style transformation to rewrite input formula into a conjunction of constraints
  - $n$-ary disjunctions of bounds (‘clauses’)
  - Arithmetic constraints having at most one operation symbol

- Boolean variables are regarded as 0-1 integer variables. Allows identification of literals with bounds on Booleans
  
  \[ b \equiv b \geq 1 \]
  
  \[ \neg b \equiv b \leq 0 \]

- Auxiliary variables $h_1, h_2, h_3$ are used for decomposition of complex constraint $x^2 - 2y \geq 6.2$. 

\[ c_1 : (\neg a \lor \neg c \lor d) \]
\[ c_2 : \land (\neg a \lor \neg b \lor c) \]
\[ c_3 : \land (\neg c \lor \neg d) \]
\[ c_4 : \land (b \lor x \geq -2) \]
\[ c_5 : \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \]
\[ c_6 : \land h_1 = x^2 \]
\[ c_7 : \land h_2 = -2 \cdot y \]
\[ c_8 : \land h_3 = h_1 + h_2 \]

\[ \text{VTSA’15 Tobias Schubert – SAT-based Test & Verification} \]
## Satisfiability Modulo Theory – iSAT

### Constraints (c_i):

- **c_1**: \( (-a \lor -c \lor d) \)
- **c_2**: \( (\neg a \lor -b \lor c) \)
- **c_3**: \( (\neg c \lor -d) \)
- **c_4**: \( (b \lor x \geq -2) \)
- **c_5**: \( (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \)
- **c_6**: \( h_1 = x^2 \)
- **c_7**: \( h_2 = -2 \cdot y \)
- **c_8**: \( h_3 = h_1 + h_2 \)

### Desired Logic (DL_1):

\( a \geq 1 \)
Satisfiability Modulo Theory – iSAT

\[ a = \text{true} \]

\[ b = \text{true} \]

### Constraints

| \( c_1 \) | \( \neg a \lor \neg c \lor d \) |
| \( c_2 \) | \( \land (\neg a \lor \neg b \lor c) \) |
| \( c_3 \) | \( \land (\neg c \lor \neg d) \) |
| \( c_4 \) | \( \land (b \lor x \geq -2) \) |
| \( c_5 \) | \( \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \) |
| \( c_6 \) | \( \land h_1 = x^2 \) |
| \( c_7 \) | \( \land h_2 = -2 \cdot y \) |
| \( c_8 \) | \( \land h_3 = h_1 + h_2 \) |
Satisfiability Modulo Theory – iSAT

\[ (\neg a \lor \neg c \lor d) \]

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

\[ \land (\neg c \lor \neg d) \]

\[ \land (b \lor x \geq -2) \]

\[ \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \]

\[ \land h_1 = x^2 \]

\[ \land h_2 = -2 \cdot y \]

\[ \land h_3 = h_1 + h_2 \]

\[ \land (\neg a \lor \neg c) \]
Satisfiability Modulo Theory – iSAT

\[ \neg a \lor \neg c \lor d \]
\[ \land (\neg a \lor \neg b \lor c) \]
\[ \land (\neg c \lor \neg d) \]
\[ \land (b \lor x \geq -2) \]
\[ \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \]
\[ \land h_1 = x^2 \]
\[ \land h_2 = -2 \cdot y \]
\[ \land h_3 = h_1 + h_2 \]
\[ \land (\neg a \lor \neg c) \]

DL 1:

- \( a \geq 1 \)
- \( c \leq 0 \)
- \( b \leq 0 \)
- \( x \geq -2 \)
Satisfiability Modulo Theory – iSAT

c₁ : \((-a \lor \neg c \lor d)\)
c₂ : \(\land (-a \lor \neg b \lor c)\)
c₃ : \(\land (-c \lor \neg d)\)
c₄ : \(\land (b \lor x \geq -2)\)
c₅ : \(\land (x \geq 4 \lor y \leq 0 \lor h₃ \geq 6.2)\)
c₆ : \(\land h₁ = x²\)
c₇ : \(\land h₂ = -2 \cdot y\)
c₈ : \(\land h₃ = h₁ + h₂\)
c₉ : \(\land (\neg a \lor \neg c)\)

DL 1:
- \(a \geq 1\)
- \(c \leq 0\)
- \(b \leq 0\)
- \(x \geq -2\)

DL 2:
- \(y \geq 4\)
- \(h₂ \leq -8\)
Satisfiability Modulo Theory – iSAT

\[c_1 : (\neg a \lor \neg c \lor d)\]
\[c_2 : \land (\neg a \lor \neg b \lor c)\]
\[c_3 : \land (\neg c \lor \neg d)\]
\[c_4 : \land (b \lor x \geq -2)\]
\[c_5 : \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2)\]
\[c_6 : \land h_1 = x^2\]
\[c_7 : \land h_2 = -2 \cdot y\]
\[c_8 : \land h_3 = h_1 + h_2\]
\[c_9 : \land (\neg a \lor \neg c)\]
Satisfiability Modulo Theory – iSAT

\[ c_1 : \quad (\neg a \lor \neg c \lor d) \]
\[ c_2 : \quad \land (\neg a \lor \neg b \lor c) \]
\[ c_3 : \quad \land (\neg c \lor \neg d) \]
\[ c_4 : \quad \land (b \lor x \geq -2) \]
\[ c_5 : \quad \land (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \]
\[ c_6 : \quad \land h_1 = x^2 \]
\[ c_7 : \quad \land h_2 = -2 \cdot y \]
\[ c_8 : \quad \land h_3 = h_1 + h_2 \]
\[ c_9 : \quad \land (\neg a \lor \neg c) \]
\[ c_{10} : \quad \land (x < -2 \lor y < 4 \lor x > 3) \]

\[ \leftarrow \text{Conflict clause = symbolic description} \]
\[ \text{of a rectangular region of the search space} \]
\[ \text{which is excluded from future search} \]

\[ \text{DL 1:} \quad \begin{array}{c}
a \geq 1 \\
\land \quad c \leq 0 \\
\land \quad b \leq 0 \quad c_4 \\
\end{array} \]
\[ \text{DL 2:} \quad \begin{array}{c}
y \geq 4 \\
\land \quad h_2 \leq -8 \quad c_7 \\
\end{array} \]
\[ \text{DL 3:} \quad \begin{array}{c}
x \leq 3 \\
\land \quad h_3 \geq 6.2 \\
\land \quad h_2 \geq -2.8 \\
\end{array} \]

\[ \text{VTSA'15 Tobias Schubert – SAT-based Test & Verification 150 / 192} \]
c_1 : \ (\neg a \lor \neg c \lor d) \\
c_2 : \ \land \ (\neg a \lor \neg b \lor c) \\
c_3 : \ \land \ (\neg c \lor \neg d) \\
c_4 : \ \land \ (b \lor x \geq -2) \\
c_5 : \ \land \ (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \\
c_6 : \ \land \ h_1 = x^2 \\
c_7 : \ \land \ h_2 = -2 \cdot y \\
c_8 : \ \land \ h_3 = h_1 + h_2 \\
c_9 : \ \land \ (\neg a \lor \neg c) \\
c_{10} : \ \land \ (x < -2 \lor y < 4 \lor x > 3)
Satisfiability Modulo Theory – iSAT

\[ \begin{align*}
\text{c}_1 & : (\neg a \lor \neg c \lor d) \\
\text{c}_2 & : (\neg a \lor \neg b \lor c) \\
\text{c}_3 & : (\neg c \lor \neg d) \\
\text{c}_4 & : (b \lor x \geq -2) \\
\text{c}_5 & : (x \geq 4 \lor y \leq 0 \lor h_3 \geq 6.2) \\
\text{c}_6 & : h_1 = x^2 \\
\text{c}_7 & : h_2 = -2 \cdot y \\
\text{c}_8 & : h_3 = h_1 + h_2 \\
\text{c}_9 & : (\neg a \lor \neg c) \\
\text{c}_{10} & : (x \leq -2 \lor y < 4 \lor x > 3)
\end{align*} \]

\[ \begin{align*}
\text{DL 1:} & \\
& a \geq 1 \quad \text{c}_9 \\
& c \leq 0 \quad \text{c}_2 \\
& b \leq 0 \quad \text{c}_4 \\
& x \geq -2 \\
\text{DL 2:} & \\
& y \geq 4 \quad \text{c}_7 \\
& h_2 \leq -8 \quad \text{c}_6 \\
& x > 3 \quad \text{c}_6 \\
& h_1 > 9
\end{align*} \]

- Continue do split and deduce until either
  - formula turns out to be UNSAT (unresolvable conflict),
  - formula turns out to be SAT (point interval),
  - solver is left with ‘sufficiently small’ portion of the search space for which it cannot derive any contradiction.

- Avoid infinite splitting and deduction
  - Minimal splitting width
  - Discard a deduced bound if it yields small progress only
Satisfiability Modulo Theory – iSAT

Remarks

- All variables have to be bounded initially
- Reliable results due to outward rounding
- Further features
  - Clever normalization rules
  - Continue search after “unknown”
  - Proof of unsatisfiability
  - Unbounded model checking using interpolants
  - Handling of stochastic constraint systems
  - Parallelization based on message passing
Example: Train Separation in Absolute Braking Distance

- Part of the forthcoming European Train Control Standard
- Minimal distance between two trains equals braking distance plus safety margin
- First train reports position of its end to the second train every 8 seconds
- Controller of the second train automatically initiates braking to maintain safety margin

Top-level view of the Matlab/Simulink model for two trains
Hybrid System Verification

Example: Train Separation in Absolute Braking Distance

- Model of controller and train dynamics

- Safety property to be checked:
  Does the controller guarantee that collisions aren’t possible?
Hybrid System Verification

Example: Train Separation in Absolute Braking Distance

```plaintext
-- Switch block: Passes through the first input or the third input
-- based on the value of the second input.

brake -> a = a_brake;
!brake -> a = a_free;
```
Hybrid System Verification

Example: Train Separation in Absolute Braking Distance

-- Relay block: When the relay is on, it remains on until the input
-- drops below the value of the switch off point parameter. When the
-- relay is off, it remains off until the input exceeds the value of
-- the switch on point parameter.

(!is_on and h \geq param_on) \rightarrow (\text{is_on'} \land \text{brake});
(!is_on and h < param_on) \rightarrow (!\text{is_on'} \land \neg \text{brake});
(\text{is_on} \land h \leq param_off) \rightarrow (!\text{is_on'} \land \neg \text{brake});
(\text{is_on} \land h > param_off) \rightarrow (\text{is_in'} \land \text{brake});
Example: Train Separation in Absolute Braking Distance

-- Euler approximation of integrator block

\[ x_r' = x_r + dt \times v; \]
Hybrid System Verification

Example: Train Separation in Absolute Braking Distance

From top to bottom positions, accelerations, speeds, and distances of the two trains are shown.
Outline

Applications

- Bounded Model / Property Checking
- Test Pattern Relaxation
- Black Box Verification
- Path Compaction
- Automatic Test Pattern Generation
- Combinational Equivalence Checking
- Security Issues
- Hybrid System Verification
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT
MaxSAT in a Nutshell

Max-SAT
- Given a CNF $\varphi$, find a truth assignment for all variables that satisfies the maximum number of clauses within $\varphi$

Variants of Max-SAT
- Partial Max-SAT
  - $\varphi$ consists of hard and soft clauses
  - All hard clauses must be satisfied
  - Maximize number of satisfied soft clauses
- Weighted Max-SAT
- Weighted Partial Max-SAT
MaxSAT in a Nutshell

Solving (Partial) Max-SAT using SAT Algorithms

- Each soft clause gets extended by a fresh “trigger” variable: 
  \((x_1 \lor x_2) \rightsquigarrow (t_1 \lor x_1 \lor x_2)\)

- By construction, after adding trigger variables all soft clauses can be satisfied simultaneously

- Now, Max-SAT corresponds to minimizing \(k\) in \(\sum_{c=1}^{m} t_c \leq k\) with \(m\) representing the number of soft clauses

- Encode \(\sum_{c=1}^{m} t_c \leq k\) with a bitonic sorting network (unary representation), convert it to CNF, and add it to the formula

- Solve the Max-SAT problem by using incremental SAT solving, iterating over \(k\)
Each arrow in the example above represents a comparator (half adder):

\[ \text{comp}(x_1, x_2, y_1, y_2) \leftrightarrow ((y_1 \leftrightarrow x_1 \lor x_2) \land (y_2 \leftrightarrow x_1 \land x_2)) \]

Using Tseitin encoding each comparator can be modeled with 2 auxiliary variables & 6 clauses
Path Compaction

- Production of circuits is erroneous
  - Various types and sources of faults
  - Covered here: Small-delay faults
Path Compaction

Sensitizable Paths and Small Delay Faults

- **Sensitizable path**: Transition from input to output
- **Length of a path according to sum of gate delays**
Path Compaction

Sensitizable Paths and Small Delay Faults

- **Small delay faults**: Assume additional delay for one gate
- Output transition too late for clock
- The longer the path the higher the detection quality
- Two-pattern delay test

![Diagram showing sensitizable paths and small delay faults with additional delay of $\delta = 2$.]
Path Compaction

- Production of circuits is erroneous
  - Various types and sources of faults
  - Covered here: Small-delay faults

- General workflow
  - Predefined paths obtained from path analysis tool
  - Sensitize all target paths using as less patterns as possible to reduce overall test overhead
  - Test pattern relaxation

- Approach
  - SAT-based maximization of sensitized target paths
Path Compaction

Maximization of Sensitized Target Paths using Partial Max-SAT

- $s^{P_i}$ indicates whether a path $p$ is sensitized or not
- $<s^{P_i}, \ldots, s^{P_n}>$ gets sorted by 1’s and 0’s
- $<SO_1, \ldots, SO_n> = <1, \ldots, 1, 0, \ldots, 0>$
- Setting $SO_i$ to 1 forces the solver to sensitize at least $i$ paths
Path Compaction

- Production of circuits is erroneous
  - Various types and sources of faults
  - Covered here: Small-delay faults

- General workflow
  - Predefined paths obtained from path analysis tool
  - Sensitize all target paths using as less patterns as possible to reduce overall test overhead
  - Test pattern relaxation

- Approach
  - SAT-based maximization of sensitized target paths

- Results
  - Applicable to large industrial circuits
  - Significantly reduced number of test patterns compared to other state-of-the-art approaches
Outline

Applications

Bounded Model / Property Checking
Path Compaction
Security Issues

Test Pattern Relaxation
Automatic Test Pattern Generation
Hybrid System Verification

Black Box Verification
Combinational Equivalence Checking
The End

Core Algorithms

SAT
MaxSAT
#SAT
QBF
DQBF
SMT
QBF in a Nutshell

Quantified Boolean Formula (QBF)

- Extension of SAT where the variables are either universal or existential quantified

Example

\[ \Psi = \exists x_1 \forall x_2, x_3 \exists x_4, \ldots, x_n \varphi(x_1, \ldots, x_n) \]

- Semantics (for this particular example)

\[ \Psi \text{ is satisfied iff there exists one assignment for } x_1 \text{ such that for every assignment of } x_2 \text{ and } x_3, \text{ there exists one assignment for } x_4, \ldots, x_n, \text{ such that } \varphi \text{ is satisfied} \]
Test Pattern Relaxation using QBF

Motivation

- Parts of the pattern get unspecified (don’t care) $\Rightarrow$ test cube
- Test properties still hold
- Reduced overall test overhead
- Focus of this work: Test cube generation with maximum number of don’t cares $\Rightarrow$ optimal test cube

Fault model considered here

- Again, small-delay Faults
Modeling Don’t Cares with QBF

Simulation for $B = 0$

$A = 1$
$B = 0$
$C = 1$

$D = 0$
$E = 1$
$F = 1$
$G = 1$

$⇒ F$ can be set to 1, even if $B$ is unspecified!

$⇒$ Don’t cares can be represented by $∀$ variables

$⇒ \exists \{A, C\} ∀\{B\} ∀\{D, E, F, G\} . \ φ(A, \ldots, G) \land (F)$
Test Pattern Relaxation using QBF

- Identifying small-delay faults requires two timeframes
- Test cube with **maximum number** of unspecified inputs using QBF
- Quantify unspecified inputs universally, specified ones existentially
- If a path for small-delay fault is sensitizeable:
  - **Universally quantified inputs**: Excluded from test cube
  - **Existential quantified inputs**: Test cube
- **But**: The quantifier of a variable **cannot be changed** in QBF
  ⇒ Unspecified inputs are not known a-priori
  ⇒ Which inputs have to be quantified universally?
**Test Pattern Relaxation using QBF**

\[
\psi = \exists SO_1, \ldots, SO_n, S_1, \ldots, S_n, E_1, \ldots, E_n \forall A_1, \ldots, A_n \exists \ldots \varphi_{circ.} \land \varphi_{prop.} \land \varphi_{mux} \land \varphi_{bsn} \land SO_k
\]

- **Dynamic** choice of (un-)specified inputs using multiplexers
- Select input \( S_i \) switches between specified (\( S_i = 0 \leadsto \exists E_i \)) and unspecified (\( S_i = 1 \leadsto \forall A_i \)) for any primary input \( I_i \)
- Find the maximum number of multiplexer select inputs that can be set to 1
- **Search for** \( k \), such that: Path is sensitizable with \( k \) unspecified inputs (\( SO_k = 1 \)), but not with \( k + 1 \) (\( SO_{k+1} = 0 \))
  \[ \implies \text{Optimal test cube, i.e., maximum number of don't cares} \]
Outline

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT
Motivation – Equivalence Checking

Are implementation and specification equivalent?
Motivation – **Partial** Equivalence Checking

Realizability, i.e. are there implementations of the black boxes (BBs) such that implementation and specification are equivalent?
QBF vs. Dependency-QBF (DQBF)

\[
\begin{align*}
\text{Specification} & \equiv \text{Implementation} \\
\equiv 1?
\end{align*}
\]

Expressible with QBF
QBF vs. Dependency-QBF (DQBF)

- Expressible with QBF
  ⇒ Approximation
  - BBs read all inputs
QBF vs. Dependency-QBF (DQBF)

- Expressible with **QBF**
  - Approximation
  - BBs read *all* inputs

- Expressible with **DQBF**
  - More precise
  - BBs read *actual* inputs
QBF vs. DQBF

QBF
- Linear quantifier-order
- Existentially quantified variables depend on all universally quantified variables left of it

\[
\psi_{QBF} = \forall x_1 \forall x_2 \exists y_1 \exists y_2 : \varphi
\]

DQBF
- Non-linear quantifier-order
- Dependencies between variables are explicitly expressible

\[
\psi_{DQBF} = Q \left( \forall x_1 \forall x_2 \exists y_1 \{ x_1 \} \exists y_2 \{ x_2 \} : \varphi \right)
\]

dependencies
Semantics of DQBF

\[ \psi_{DQBF} = \forall x_1 \forall x_2 \exists y_1 \{ x_1 \} \exists y_2 \{ x_2 \} : \varphi \]

Additional constraints compared to QBF

1) For the same assignment of all \( \forall \) variables \( u \in \text{dep}(e) \) the assignment of the \( \exists \) variable \( e \) has to be the same

2) For different assignments of at least one \( \forall \) variable \( u \in \text{dep}(e) \) the assignment of the \( \exists \) variable \( e \) is allowed to change
QBF and DQBF for Partial Equivalence Checking

QBF
- Does not take dependencies between BBs into account
- BBs read all circuit inputs
- UNSAT $\Rightarrow$ unrealizability
- SAT $\Leftrightarrow$ realizability

DQBF
- BBs read only affecting signals
- UNSAT $\Rightarrow$ unrealizability
- SAT $\Rightarrow$ realizability

For one black box QBF is as accurate as DQBF!
DQBF-based Partial Equiv. Checking – Example

\[ \phi = (y_1 + y_2) \oplus (x_1 \oplus x_2) \]

<table>
<thead>
<tr>
<th>a (\oplus) b</th>
<th>a (\oplus) b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \forall x_1 \forall x_2 \exists y_1 \exists y_2 : \phi \]

\[ \forall x_2 \exists y_2 \forall x_1 \exists y_1 : \phi \]

\[ \forall x_1 \forall x_2 \exists y_1 \exists y_2 : \phi \]

\[ QBF \text{ Approx.} \]
∀x_1 ∀x_2 ∃y_1 ∃y_2 : (y_1 + y_2) ⊕ (x_1 ⊕ x_2)

⇒ SAT! ⇒ Impl. Realizable!

WRONG!
∀x₁ ∀x₂ ∃y₁(x₁) ∃y₂(x₂) : (y₁ + y₂) ⊕ (x₁ ⊕ x₂)
∀x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : (y_1 + y_2) \oplus (x_1 \oplus x_2)

<table>
<thead>
<tr>
<th>y_1</th>
<th>y_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1 = 0 \rightarrow y_1 = 0</td>
<td></td>
</tr>
</tbody>
</table>

1 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0
∀x_1 ∀x_2 ∃y_1(x_1) ∃y_2(x_2) : (y_1 + y_2) ⊕ (x_1 ⊕ x_2)

<table>
<thead>
<tr>
<th>y_1</th>
<th>y_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x_1 : 0 → y_1 = 0</td>
<td>x_2 : 0 → y_2 = 0</td>
</tr>
</tbody>
</table>
DQBF-based Partial Equiv. Checking – Example

\[ \forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : (y_1 + y_2) \oplus (x_1 \oplus x_2) \]

<table>
<thead>
<tr>
<th></th>
<th>( y_1 )</th>
<th>( y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 = 0 )</td>
<td>( y_1 = 0 )</td>
<td>( x_2 = 0 )</td>
</tr>
</tbody>
</table>

1  0  0  0  0  1  1  1  0  1  1  1  1  0  0  0
DQBF-based Partial Equiv. Checking – Example

∀x₁ ∀x₂ ∃y₁(x₁) ∃y₂(x₂) : (y₁ + y₂) ⊕ (x₁ ⊕ x₂)

<table>
<thead>
<tr>
<th>y₁</th>
<th>y₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

x₁ = 0 → y₁ = 0
x₂ = 0 → y₂ = 0
\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) : (y_1 + y_2) \oplus (x_1 \oplus x_2)

<table>
<thead>
<tr>
<th></th>
<th>y_1</th>
<th>y_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1 = 0 \rightarrow y_1 = 0</td>
<td>y_2 = 0 \rightarrow y_2 = 0</td>
<td></td>
</tr>
<tr>
<td>x_2 = 1 \rightarrow y_2 = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
∀ x_1 ∀ x_2 ∃ y_1(x_1) ∃ y_2(x_2) : (y_1 + y_2) ⊕ (x_1 ⊕ x_2)

<table>
<thead>
<tr>
<th></th>
<th>y_1</th>
<th>y_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1 = 0 → y_1 = 0</td>
<td>x_2 = 0 → y_2 = 0</td>
<td></td>
</tr>
<tr>
<td>x_2 = 1 → y_2 = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
∀x₁ ∀x₂ ∃y₁(x₁) ∃y₂(x₂) : (y₁ + y₂) ⊕ (x₁ ⊕ x₂)

<table>
<thead>
<tr>
<th></th>
<th>y₁</th>
<th>y₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁ = 0 → y₁ = 0</td>
<td></td>
<td>x₂ = 0 → y₂ = 0</td>
</tr>
<tr>
<td>x₁ = 1 → y₁ = 0</td>
<td></td>
<td>x₂ = 1 → y₂ = 1</td>
</tr>
</tbody>
</table>

1 0 0 0 0 1 1 1 0 0 0 0
∀x₁ ∀x₂ ∃y₁(x₁) ∃y₂(x₂) : (y₁ + y₂) ⊕ (x₁ ⊕ x₂)

<table>
<thead>
<tr>
<th></th>
<th>y₁</th>
<th>y₂</th>
</tr>
</thead>
</table>
| x₁ = 0 → y₁ = 0 | x₂ = 0 → y₂ = 0
| x₁ = 1 → y₁ = 0 | x₂ = 1 → y₂ = 1

\[ \forall x₁ \forall x₂ \exists y₁(x₁) \exists y₂(x₂) : (y₁ + y₂) \oplus (x₁ \oplus x₂) \]
∀x₁ ∀x₂ ∃y₁(x₁) ∃y₂(x₂) : (y₁ + y₂) ⊕ (x₁ ⊕ x₂)

\begin{array}{|c|c|}
\hline
y₁ & y₂ \\
\hline
x₁ = 0 → y₁ = 0 & x₂ = 0 → y₂ = 0 \\
x₁ = 1 → y₁ = 0 & x₂ = 1 → y₂ = 1 \\
\hline
\end{array}

⇒ \text{UNSAT} ⇒ \text{CORRECT}
Henkin Quantified Solver (HQS)

Gate detection
- Generate dependency graph $G_\phi$
- Transform $\phi$ into AIG

Preprocessing
- Eliminate universal
- Eliminate unit/pure
- Eliminate existential

DQBF $\phi$
- Choose next universal

Elimination Loop
- $G_\phi$ acyclic?
  - Yes: Apply QBF solver
  - No

(Un-)satisfiable
Main Idea behind HQS – Acyclic Dependency Graph

There is an edge from $a$ to $b$, iff:

There is an edge from $a$ to $b$, iff:

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2)$$

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_1, x_2)$$

$a$ depends on variables, on which $b$ does not.

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_1, x_2) = \forall x_1 \exists y_1 \forall x_2 \exists y_2$$

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_1, x_2)$$

acyclic $\rightarrow$ DQBF $\equiv$ QBF
Outline

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT
#SAT in a Nutshell

#SAT

- Given a CNF $\varphi$, count how many disjoint truth assignments satisfy $\varphi$.
- #SAT solver have to continue search after one solution has been found.
- With $n$ variables, $\varphi$ can have up to $2^n$ satisfying assignments.
- #SAT corresponds to model counting, not enumerating all satisfying assignments.
- Accelerating techniques differ from classical SAT solving:
  - Caching of already analyzed sub-formulae: $[\varphi', M_{\varphi'}]$.
  - Component analysis: $\varphi = \varphi' \land \varphi'' \Rightarrow M_\varphi = M_{\varphi'} \cdot M_{\varphi''}$.
- Different approaches: Exact vs. approximate model counting.
#SAT – Example

\[
\varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5)
\]
\[ \varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5) \]

\[ (v_1 \lor \neg v_2) \land (v_1 \lor v_2) \land (\neg v_4 \lor v_5) \]

\[ (\neg v_2) \land (v_2) \land (\neg v_4 \lor v_5) \]
#SAT – Example

\[ \varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5) \]
ϕ = (υ₁ ∨ ¬υ₂) ∧ (υ₁ ∨ υ₂ ∨ υ₃) ∧ (¬υ₄ ∨ υ₅) ∧ (¬υ₃ ∨ υ₅)
\[ \varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5) \]
\[ \varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5) \]
\( \varphi = (v_1 \lor \neg v_2) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_4 \lor v_5) \land (\neg v_3 \lor v_5) \)

\[\text{mc}(\varphi) = 12\]
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice

\[ \varphi = (v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor v_3) \]
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- \( \varphi = (v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor v_3) \)
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice

\[ \varphi = (v_1 \vee v_2 \vee v_3) \land (\neg v_1 \vee v_2 \vee v_3) \]

Diagram:

- Node 1: \((v_3)\) (false)
- Node 2: \((v_2 \vee v_3)\) (true)
- Node 3: \((v_2 \vee v_3)\) (false)
- Node \(v_1\) with edges to nodes 1, 2, and 3
- Edge from node 1 to node \(v_1\) labeled "false"
- Edge from node 3 to node \(v_1\) labeled "false"
- Edge from node 2 to node \(v_1\) labeled "true"

Cache hit from node 2 to node 1
Store model counts of sub-formulas in a cache

Do not compute the result for the same sub-formula twice

\( \varphi = (v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor v_3) \)
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- \( \varphi = (v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor v_3) \)
The formula might split into disjoint sub-formulas
The formula might split into disjoint sub-formulas

\[ \phi = (\neg p_2 \lor a_2) \land (a_1 \lor a_2 \lor a_3) \land (b_1) \land (\neg b_3 \lor b_4) \land (p_2 \lor \neg b_2) \]
The formula might split into disjoint sub-formulas

\[ \varphi = (\neg p_2 \lor a_2) \land (a_1 \lor a_2 \lor a_3) \land (b_1) \land (\neg b_3 \lor b_4) \land (p_2 \lor \neg b_2) \]

Assignment: \( p_2 = \text{false} \)
The formula might split into disjoint sub-formulas

\[ \varphi = (\neg p_2 \lor a_2) \land (a_1 \lor a_2 \lor a_3) \land (b_1) \land (\neg b_3 \lor b_4) \land (p_2 \lor \neg b_2) \]

Assignment: \( p_2 = false \)

Sub-formulas:

\[ \varphi_1 = (a_1 \lor a_2 \lor a_3) \]

\[ \varphi_2 = (b_1) \land (\neg b_3 \lor b_4) \land (\neg b_2) \]
The formula might split into disjoint sub-formulas

\[ \varphi = (\neg p_2 \lor a_2) \land (a_1 \lor a_2 \lor a_3) \land (b_1) \land (\neg b_3 \lor b_4) \land (p_2 \lor \neg b_2) \]

Assignment: \( p_2 = \text{false} \)

Sub-formulas:

- \( \varphi_1 = (a_1 \lor a_2 \lor a_3) \)
- \( \varphi_2 = (b_1) \land (\neg b_3 \lor b_4) \land (\neg b_2) \)

Model count is computed by multiplying results for sub-formulas:

\[ mc(\varphi|_{p_2=\text{false}}) = mc(\varphi_1) \cdot mc(\varphi_2) = 7 \cdot 3 = 21 \]
Security Issues – Fault Injection

- Extract secret information from a security circuit (AES, …)
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret
Security Issues – Fault Injection

- Extract secret information from a security circuit (AES, …)
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret

![Security Circuit Diagram]

Security circuit

Clock

Input

Output

Combinational circuit

Flip-Flops
Security Issues – Fault Injection

- Extract secret information from a security circuit (AES, … )
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret

![Security Circuit Diagram]

Flip-flops store value on rising clock edge
Successful injection: flip-flops store an incorrect value
How likely is a successful injection for unknown input?
Security Issues – Fault Injection

- Extract secret information from a security circuit (AES, …)
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret

- Flip-flops store value on rising clock edge
Security Issues – Fault Injection

- Extract secret information from a security circuit (AES, …)
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret

- Flip-flops store value on rising clock edge
- Successful injection: flip-flops store an incorrect value
- How likely is a successful injection for unknown input?
Security Issues – Fault Injection

1. Encode combinational circuit and its timing as CNF formula $\varphi$ with the tool WaveSAT$^1$
2. Make $\varphi$ satisfiable iff at least one fault is injected
3. Add conditions for outputs that must be correct

Security Issues – Fault Injection

1. Encode combinational circuit and its timing as CNF formula $\phi$ with the tool WaveSAT$^1$

2. Make $\phi$ satisfiable iff at least one fault is injected

3. Add conditions for outputs that must be correct

4. Calculate number of satisfying assignments $mc(\phi)$

5. $P(\text{Successful Injection}) = \frac{mc(\phi)}{2^{\#\text{circuit inputs}}}$

---

Conclusion

Applications

- Bounded Model / Property Checking
- Path Compaction
- Security Issues
- Test Pattern Relaxation
- Automatic Test Pattern Generation
- Hybrid System Verification
- Black Box Verification
- Combinational Equivalence Checking
- The End

Core Algorithms

- SAT
- MaxSAT
- #SAT
- QBF
- DQBF
- SMT

VTSA’15 Tobias Schubert – SAT-based Test & Verification 191 / 192
Some Papers...


[Lewis, Marin, Schubert, Narizzano, Becker, Giunchiglia. *Parallel QBF Solving with Advanced Knowledge Sharing*. Fundamenta Informaticae, 2011]


