SAT-based Approaches for Test & Verification of Integrated Circuits (Part II)

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert

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

Summary School on Verification Technology, Systems & Applications 2015

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

UNI FREIBURG

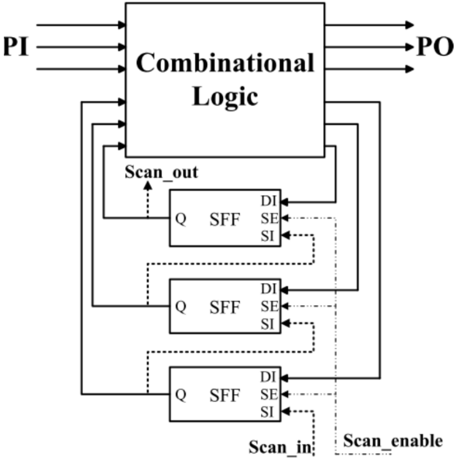
SAT-based ATPG – Testing of Sequential Circuits

Problems specific wrt. test of sequential circuits

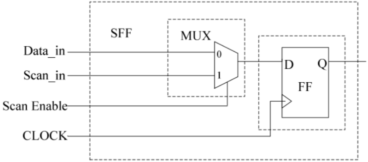
- Initialization
  - Circuit's state at the beginning of test application might be unknown
- Counters
  - Setting a counter to a specific value might take a lot of clock cycles
- Complexity of test generation
  - Finding a sequence to distinguish between a faulty and a fault-free chip might require a large number of state transitions

$\Rightarrow$ Practical methods reduce sequential to combinatorial ATPG
$\Rightarrow$ Solution: "Design for Testability"-techniques within the chips
$\Rightarrow$ Example: Scan-based designs

SAT-based ATPG – Scan-based Designs



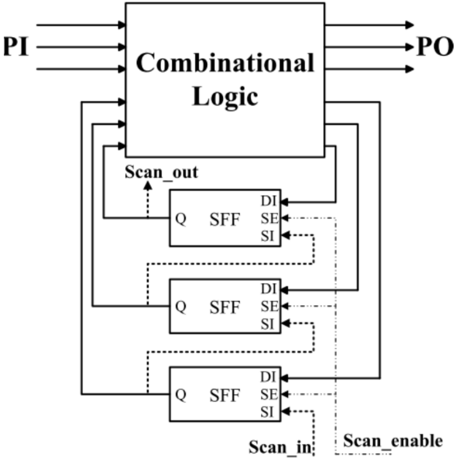
Scan Flip-Flop (SFF)



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

UNI FREIBURG

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

Outline

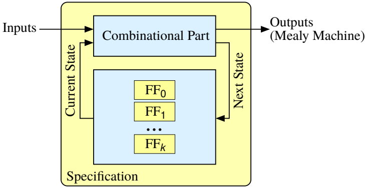
Applications

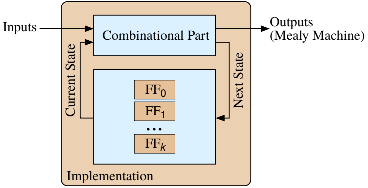
Bounded Model / Property Checking	Path Compaction	Security Issues
Test Pattern Relaxation	Automatic Test Pattern Generation	Hybrid System Verification
Black Box Verification	Combinational Equivalence Checking	The End

SAT	MaxSAT	#SAT	QBF	DQBF	SMT

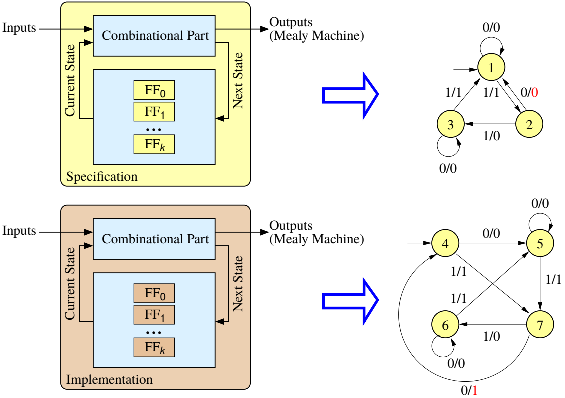
Core Algorithms

Sequential Equivalence Checking

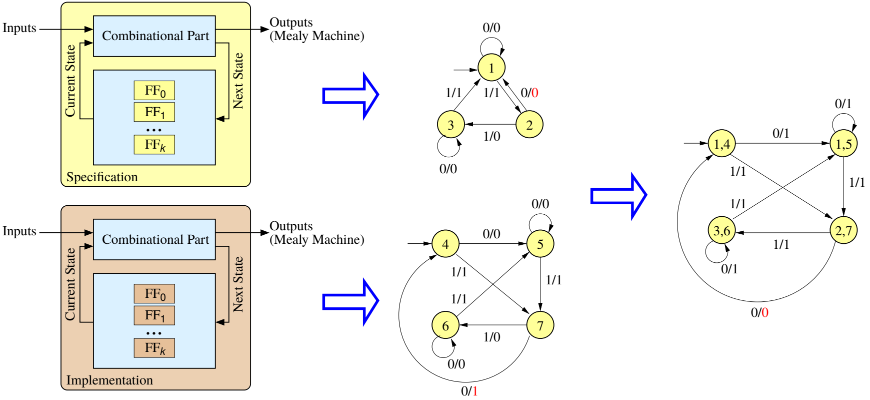




Sequential Equivalence Checking



Sequential Equivalence Checking



Sequential Equivalence Checking

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

  $\Rightarrow$ 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 $L : S \to 2^V$, where V is a set of propositional variables (atomic formulas, atomic propositions).

- Atomic propositions are observable elementary properties of states, like “a timeout has occured”, “a request has been made”
- Using such a temporal structure, we can derive all possible computational runs. They are obtained by “unrolling” the Kipke structure according to its transition relation R

Preliminaries – Temporal Propositional Logic

Temporal propositional logic = Propositional logic + Temporal operators

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:

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**φ: Formula φ holds in every state on the path ("*globally*" or "*always*")

Path quantifiers

They make statements about properties of states:

UNI FREIBURG

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**φ: Formula φ holds in every state on the path ("*globally*" or "*always*")
- **F**φ: Formula φ 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**φ: Formula φ holds in every state on the path ("*globally*" or "*always*")
- **F**φ: Formula φ holds in some state on the path ("*finally*" or "*eventually*")
- **X**φ: Formula φ 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:

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

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

Path quantifiers

They make statements about properties of states:

- $\mathbf{A}\varphi$: Formula φ holds on all paths starting in this state ("*for all paths*")

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:

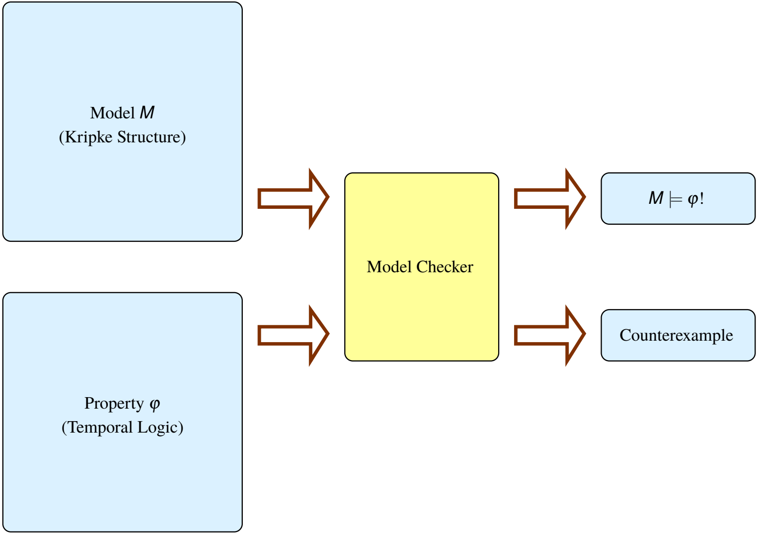
- $\mathbf{G}\varphi$: Formula φ holds in every state on the path ("*globally*" or "*always*")
- $\mathbf{F}\varphi$: Formula φ holds in some state on the path ("*finally*" or "*eventually*")
- $\mathbf{X}\varphi$: Formula φ holds in the second state on the path ("*next*")
- $\varphi\mathbf{U}\psi$: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

Path quantifiers

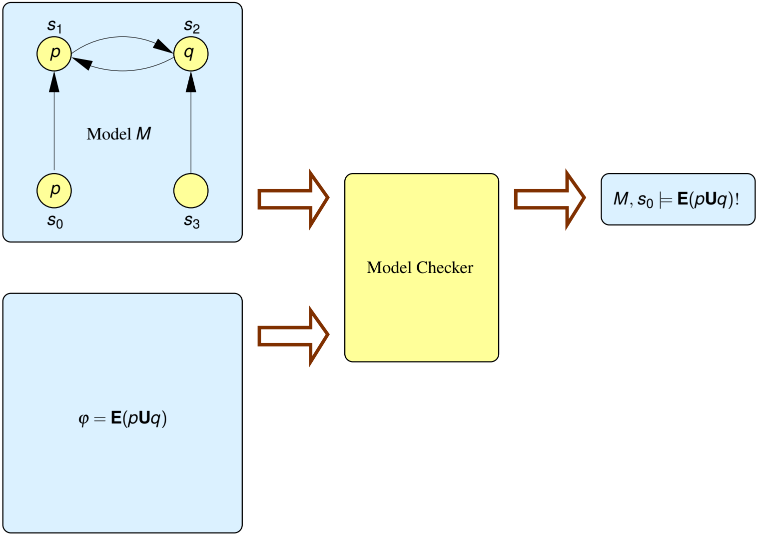
They make statements about properties of states:

- $\mathbf{A}\varphi$: Formula φ holds on all paths starting in this state ("*for all paths*")
- $\mathbf{E}\varphi$: Formula φ holds on some path starting in this state ("*there exists a path*")

Property/Model Checking in a Nutshell



Property/Model Checking in a Nutshell



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 $\Rightarrow$ bounded model checking

Model Checking vs. Bounded Model Checking

Given

- Kripke structure M
- Temporal formula φ "suited for BMC"
- Maximum unrolling depth k

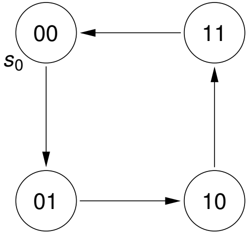
Model Checking

- $M \models \varphi$?

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



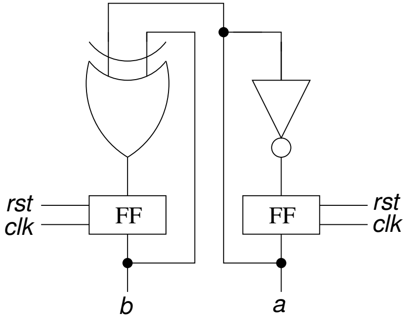
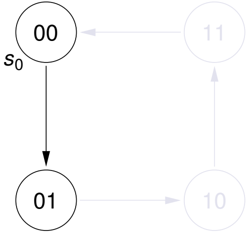
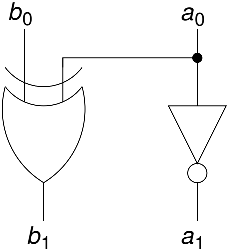


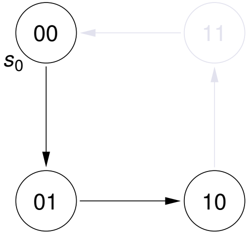
Illustration 2-Bit Counter: Time Frame Expansion

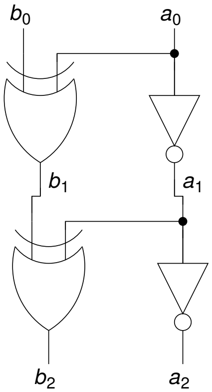




Let φ be a temporal formula and $k = 1$. $M \models_1 \varphi$?

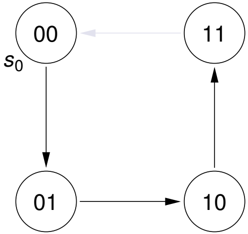
Illustration 2-Bit Counter: Time Frame Expansion

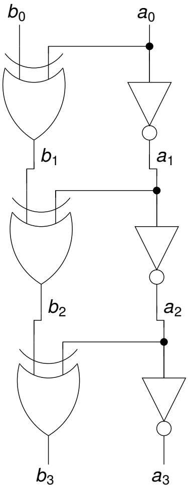




Let φ be a temporal Formula and $k = 2$. $M \models_2 \varphi$?

Illustration 2-Bit Counter: Time Frame Expansion





Let φ 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 φ, 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, φ a property, and k an unfolding depth. Then the characteristic function $[\![M, \varphi]\!]_k$ corresponding to M, φ, and k is defined as

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

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 φ at depth k.

Types of Properties – Safety

Safety

- Specify invariants of the system:

$$\mathbf{AG}\, safe$$

- BMC-formulation for refuting safety (= proving $\mathbf{EF}\neg safe$):

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg safe(s_k)$$

Types of Properties – Liveness

Liveness

- Specified in temporal logic:

$$\textbf{AF}\,\mathit{good}$$

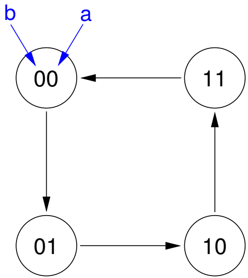
- Refutation of liveness (= proving **EG**$\neg$*good*) requires infinitely long paths!
- If **AF***good* is violated, there is a “lasso” on which all states satisfy $\neg$*good*
- BMC-formulation:

$$I(s_0) \wedge \bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k} \neg\mathrm{good}(s_i) \wedge \bigvee_{l=0}^{k} (s_l = s_{k+1})$$

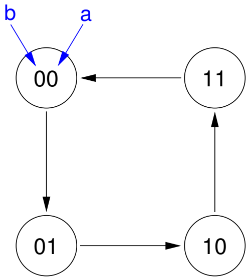
BMC Example Safety – 2-Bit Counter

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

$$\mathbf{AG}(\neg(b \wedge a)) \Leftrightarrow \neg\mathbf{EF}(b \wedge a)$$



BMC Example Safety – 2-Bit Counter

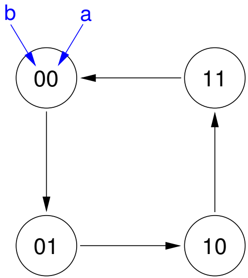


Requirement: State $(1,1)$ may not reached, or later an overflow will occur, i.e. the following must hold:

$$\mathbf{AG}(\neg(b \wedge a)) \Leftrightarrow \neg\mathbf{EF}(b \wedge a)$$

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

BMC Example Safety – 2-Bit Counter



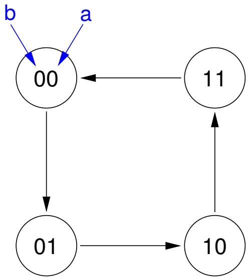
Requirement: State $(1,1)$ may not reached, or later an overflow will occur, i.e. the following must hold:

$$\mathbf{AG}(\neg(b \wedge a)) \Leftrightarrow \neg\mathbf{EF}(b \wedge a)$$

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

- $\Rightarrow$ $M \models_2 \varphi$ with $\varphi = \mathbf{EF}(b \wedge a)$?
- $\Rightarrow$ $I(s_0) = \neg b_0 \wedge \neg a_0$
- $\Rightarrow$ $R(s_0, s_1) = (b_1 \leftrightarrow (b_0 \oplus a_0)) \wedge (a_1 \leftrightarrow \neg a_0)$
- $\Rightarrow$ $R(s_1, s_2) = (b_2 \leftrightarrow (b_1 \oplus a_1)) \wedge (a_2 \leftrightarrow \neg a_1)$
- $\Rightarrow$ $P_2(\varphi) = (b_0 \wedge a_0) \vee (b_1 \wedge a_1) \vee (b_2 \wedge a_2)$
- $\Rightarrow$ $[\![M, \varphi]\!]_2 = I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge P_2(\varphi)$
- $\Rightarrow$ $[\![M, \varphi]\!]_2 = 0$
- $\Rightarrow$ Starting from $(0,0)$, $(1,1)$ cannot reached in max. 2 steps $\Rightarrow M \not\models_2 \varphi$!

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:

$$\mathbf{AG}(\neg(b \wedge a)) \Leftrightarrow \neg\mathbf{EF}(b \wedge a)$$

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

- $\Rightarrow$ $M \models_2 \varphi$ with $\varphi = \mathbf{EF}(b \wedge a)$?
- $\Rightarrow$ $I(s_0) = \neg b_0 \wedge \neg a_0$
- $\Rightarrow$ $R(s_0,s_1) = (b_1 \leftrightarrow (b_0 \oplus a_0)) \wedge (a_1 \leftrightarrow \neg a_0)$
- $\Rightarrow$ $R(s_1,s_2) = (b_2 \leftrightarrow (b_1 \oplus a_1)) \wedge (a_2 \leftrightarrow \neg a_1)$
- $\Rightarrow$ $P_2(\varphi) = (b_0 \wedge a_0) \vee (b_1 \wedge a_1) \vee (b_2 \wedge a_2)$
- $\Rightarrow$ $[\![M,\varphi]\!]_2 = I(s_0) \wedge R(s_0,s_1) \wedge R(s_1,s_2) \wedge P_2(\varphi)$
- $\Rightarrow$ $[\![M,\varphi]\!]_2 = 0$
- $\Rightarrow$ Starting from $(0,0)$, $(1,1)$ cannot reached in max. 2 steps $\Rightarrow M \not\models_2 \varphi$!

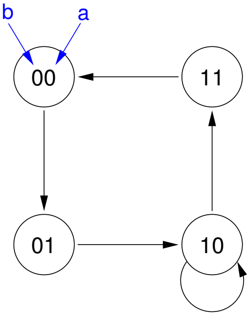
But: $M \not\models \mathbf{AG}(\neg(b \wedge a)) \Leftrightarrow M \not\models \neg\mathbf{EF}(b \wedge a)$!

UNI FREIBURG

BMC Example Liveness – Modified 2-Bit counter

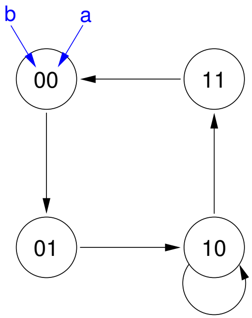
Requirement: State $(1,1)$ must be reachable from every state, i.e. the following must hold:

$$\mathbf{AF}(b \wedge a) \Leftrightarrow \neg\mathbf{EG}(\neg(b \wedge a))$$



UNI FREIBURG

BMC Example Liveness – Modified 2-Bit counter

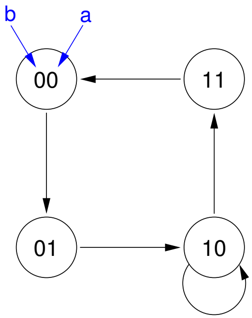


Requirement: State $(1,1)$ must be reachable from every state, i.e. the following must hold:

$$\mathbf{AF}(b \wedge a) \Leftrightarrow \neg\mathbf{EG}(\neg(b \wedge a))$$

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 $(b \wedge a)$ holds. Given $k = 2$ and $\varphi = \mathbf{EG}(\neg(b \wedge a))$:

BMC Example Liveness – Modified 2-Bit counter



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

$$\mathbf{AF}(b \wedge a) \Leftrightarrow \neg\mathbf{EG}(\neg(b \wedge a))$$

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 $(b \wedge a)$ holds. Given $k = 2$ and $\varphi = \mathbf{EG}(\neg(b \wedge a))$:

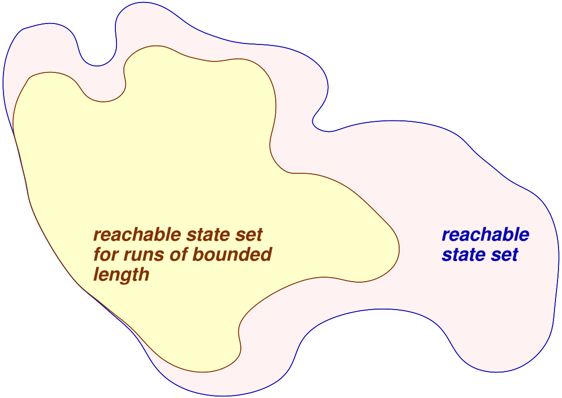
- $\Rightarrow$ $I(s_0) = \neg b_0 \wedge \neg a_0$
- $\Rightarrow$ $R(s_i, s_{i+1}) = ((b_{i+1} \leftrightarrow (b_i \oplus a_i)) \wedge (a_{i+1} \leftrightarrow \neg a_i)) \vee$ $(b_{i+1} \wedge \neg a_{i+1} \wedge b_i \wedge \neg a_i)$ with $i = 0, 1, 2$
- $\Rightarrow$ $P_2(\varphi) = (\neg b_0 \vee \neg a_0) \wedge (\neg b_1 \vee \neg a_1) \wedge (\neg b_2 \vee \neg a_2)$
- $\Rightarrow$ $[s_3 \equiv s_i] = (b_3 \leftrightarrow b_i) \wedge (a_3 \leftrightarrow a_i)$ with $i = 0, 1, 2$
- $\Rightarrow$ $[\![M, \varphi]\!]_2 = I(s_0) \wedge \left[\bigwedge_{i=0}^{2} R(s_i, s_{i+1}) \right] \wedge \left[\bigvee_{i=0}^{2} [s_3 \equiv s_i] \right] \wedge P_2(\varphi)$
- $\Rightarrow$ $[\![M, \varphi]\!]_2 = \neg b_0 \wedge \neg a_0 \wedge \neg b_1 \wedge a_1 \wedge b_2 \wedge \neg a_2 \wedge b_3 \wedge \neg a_3$
- $\Rightarrow$ Counterexample found!

SAT-based Bounded Model Checking

- BMC can be used to disprove invariants $\mathbf{AG}\varphi$
  - ... by proving $\mathbf{EF}\neg\varphi$ 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 $\mathbf{AF}\varphi$
  - ... by proving $\mathbf{EG}\neg\varphi$ 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

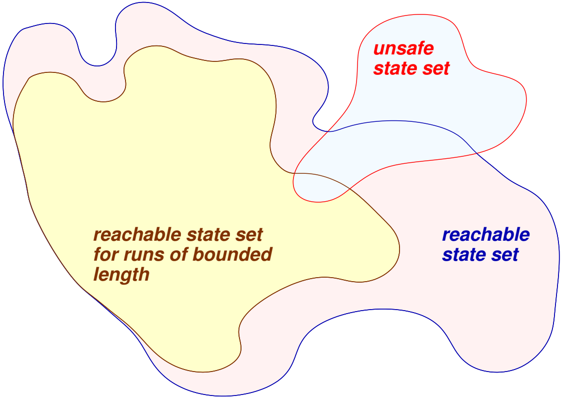
Usage of BMC to falsify Safety Properties

Idea: Restrict system behavior to runs of some given bounded length, i.e. runs with a bounded number of transition steps

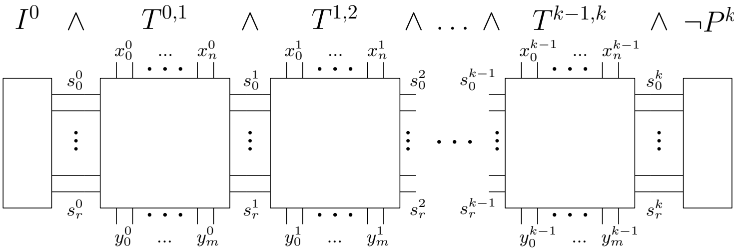


Usage of BMC to falsify Safety Properties

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 $k = 0, 1, ..., K$ up to a given maximal unrolling depth K

$$\mathrm{BMC}_k = I^0 \wedge \bigwedge_{i=0}^{k-1} T^{i,i+1} \wedge \neg P^k$$

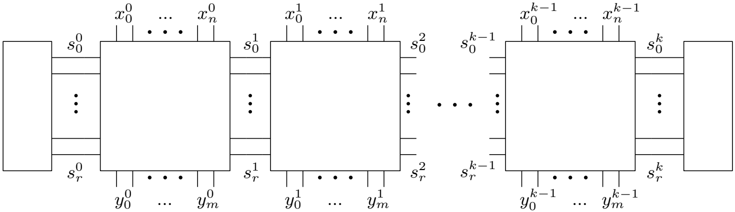
- Convert BMC_k into CNF by Tseitin transformation and solve it using a SAT solver
  - CNF satisfiable $\Rightarrow$ Invariant condition P violated after k steps
  - CNF unsatisfiable $\Rightarrow$ no conclusion, next iteration step

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 φ for a given invariant $\mathbf{AG}\varphi$?
- BMC is really efficient if there is a short error path
- Without extensions it is not possible to prove that φ holds for all reachable states
- Bounded Model Checking $\rightarrow$ Model Checking
  - Computing the "radius" of the Kripke structure
  - k-induction
  - Craig interpolation

Observation

$$I^0 \quad \wedge \quad T^{0,1} \quad \wedge \quad T^{1,2} \quad \wedge \quad \ldots \quad \wedge \quad T^{k-1,k} \quad \wedge \quad \neg P^k$$



$k = i:$ $\quad I^0 \wedge T^{0,1} \wedge T^{1,2} \wedge \ldots \wedge T^{i-1,i} \wedge \neg P^i$

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

- The main part of the formula remains unchanged
- $\neg P^i$ has to be removed
- $T^{i,i+1} \wedge \neg P^{i+1}$ 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 ($\rightsquigarrow$ 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

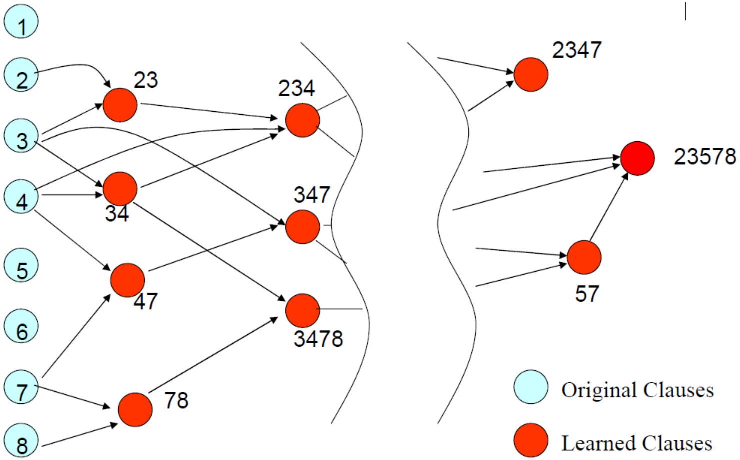


Illustration: Re-using Clauses

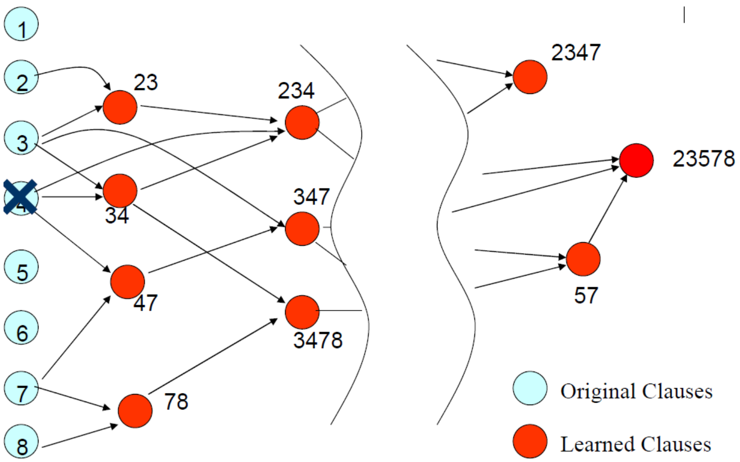
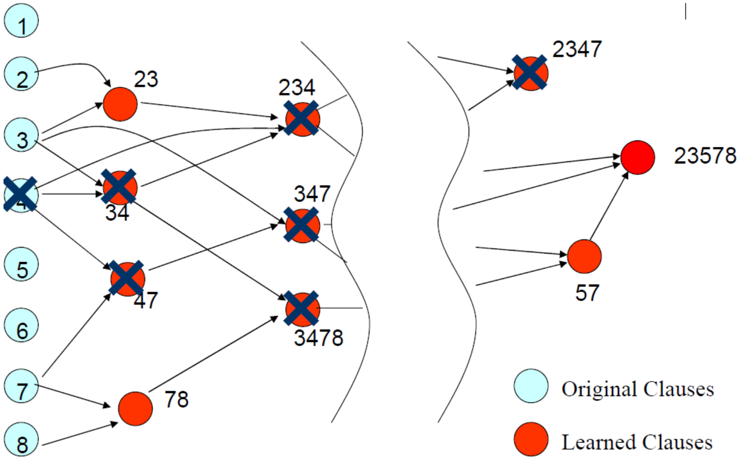


Illustration: Re-using Clauses



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 $d_i = 1$

UNI FREIBURG

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

$\varphi = (a \vee b) \wedge (\neg c \vee d)$	Initial formula
$\varphi_{0\,/\neg d_0} = (a \vee b) \wedge (\neg c \vee d) \wedge (b \vee d_0)$	incr. step 0
$\varphi_{1\,/d_0, \neg d_1} = (a \vee b) \wedge (\neg c \vee d) \wedge (b \vee d_0) \wedge (d \vee d_1)$	incr. step 1

Incremental SAT Solving with Assumptions

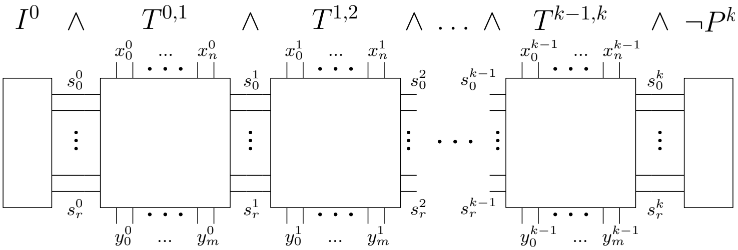
Activation variables and assumptions

- ...
- 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



$k = i:$ $\quad I^0 \wedge T^{0,1} \wedge T^{1,2} \wedge \ldots \wedge T^{i-1,i} \wedge \neg P^i$

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

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

SAT	MaxSAT	#SAT	QBF	DQBF	SMT

Core Algorithms

Satisfiability Modulo Theory

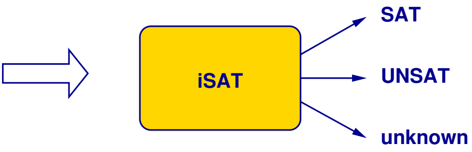
Hybrid Systems

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

iSAT

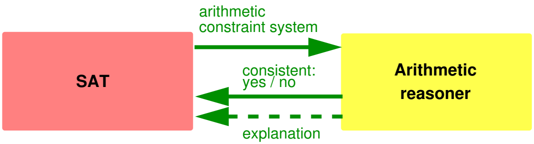
- Satisfiability and BMC checker for quantifier-free Boolean combinations of arithmetic constraints over the reals and integers

$$
\begin{aligned}
& (\neg b \vee \neg c) \\
\wedge\ & (b \rightarrow \sin(x) \cdot y < 7.2) \\
\wedge\ & (\sqrt{2x - y} = 8 \vee c) \\
\wedge\ & (i^2 = 3j - 5)
\end{aligned}
$$



iSAT

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

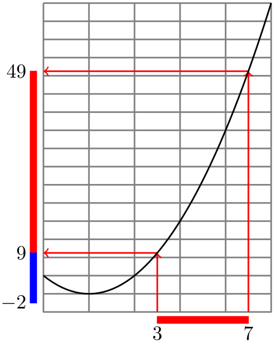


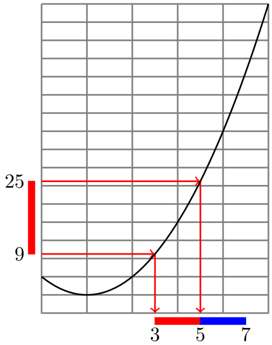
- Can be seen as a generalization of a SAT solver
  - Branch-and-deduce framework inherited from SAT
  - Deduction rule for clauses
    - Unit propagation
  - Deduction rules for arithmetic operators
    - Interval constraint propagation

Satisfiability Modulo Theory – ICP

Interval Constraint Propagation (ICP)

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

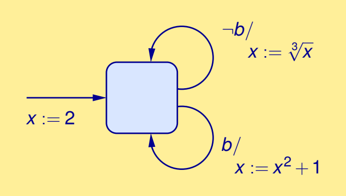




$z \in [3,7] \Rightarrow h_1 \geq 9 \Rightarrow h_1 \in [9,25]$

$h_1 \in [9,25] \Rightarrow z \leq 5 \Rightarrow z \in [3,5]$

Satisfiability Modulo Theory – BMC Mode of iSAT



Safety property:
There's no sequence of input values such that $3.14 \leq x \leq 3.15$

```
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;
```

iSAT

COUNTEREXAMPLE

```
CANDIDATE SOLUTION:
  b (boole):
   @0: [1, 1]
   @1: [0, 0]
   @2: [0, 0]
   @3: [0, 0]
   @4: [1, 1]
   @5: [1, 1]
   @6: [1, 1]
   @7: [0, 0]
   @8: [0, 0]
   @9: [1, 1]
   @10: [0, 0]
   @11: [1, 1]

  x (float):
   @0: [2, 2]
   @1: [5, 5]
   @2: [1.7099, 1,7100]
   @3: [1.1874, 1,1959]
   @4: [1.0589, 1.0615]
   @5: [2.1214, 2.1267]
   @6: [5.5013, 5.5114]
   @7: [31.329, 31.3391]
   @8: [3.1499, 1.1576]
   @9: [1.4597, 1.4671]
   @10: [3.1307, 3.1402]
   @11: [1.4629,1.4663]
   @12: [3.1400, 3.1500]
```

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

c_1 : $(\neg a \vee \neg c \vee d)$
c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$
c_3 : $\wedge\ (\neg c \vee \neg d)$
c_4 : $\wedge\ (b \vee x \geq -2)$
c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

c_6 : $\wedge\ h_1 = x^2$
c_7 : $\wedge\ h_2 = -2 \cdot y$
c_8 : $\wedge\ h_3 = h_1 + h_2$

- 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 $x^2 - 2y \geq 6.2$.

UNI FREIBURG

Satisfiability Modulo Theory – iSAT

c_1 : $(\neg a \vee \neg c \vee d)$

c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$

c_3 : $\wedge\ (\neg c \vee \neg d)$

c_4 : $\wedge\ (b \vee x \geq -2)$

c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

c_6 : $\wedge\ h_1 = x^2$

c_7 : $\wedge\ h_2 = -2 \cdot y$

c_8 : $\wedge\ h_3 = h_1 + h_2$

DL 1: $a \geq 1$

Satisfiability Modulo Theory – iSAT

$c_1: \quad (\neg a \vee \neg c \vee d)$

$c_2: \quad \wedge \ (\neg a \vee \neg b \vee c)$

$c_3: \quad \wedge \ (\neg c \vee \neg d)$

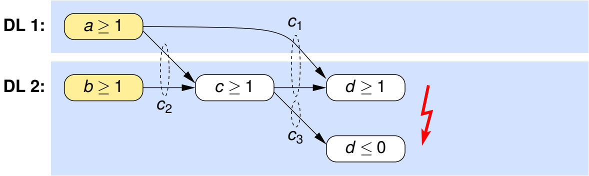
$c_4: \quad \wedge \ (b \vee x \geq -2)$

$c_5: \quad \wedge \ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

$c_6: \quad \wedge \ h_1 = x^2$

$c_7: \quad \wedge \ h_2 = -2 \cdot y$

$c_8: \quad \wedge \ h_3 = h_1 + h_2$



Satisfiability Modulo Theory – iSAT

$c_1: \quad (\neg a \vee \neg c \vee d)$

$c_2: \wedge\ (\neg a \vee \neg b \vee c)$

$c_3: \wedge\ (\neg c \vee \neg d)$

$c_4: \wedge\ (b \vee x \geq -2)$

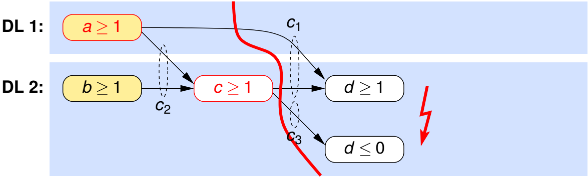
$c_5: \wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

$c_6: \wedge\ h_1 = x^2$

$c_7: \wedge\ h_2 = -2 \cdot y$

$c_8: \wedge\ h_3 = h_1 + h_2$

$c_9: \wedge\ (\neg a \vee \neg c)$



Satisfiability Modulo Theory – iSAT

c_1 : $(\neg a \vee \neg c \vee d)$

c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$

c_3 : $\wedge\ (\neg c \vee \neg d)$

c_4 : $\wedge\ (b \vee x \geq -2)$

c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

c_6 : $\wedge\ h_1 = x^2$

c_7 : $\wedge\ h_2 = -2 \cdot y$

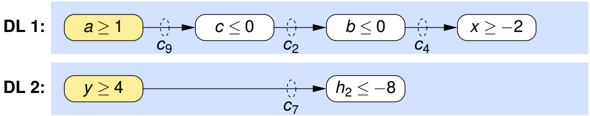
c_8 : $\wedge\ h_3 = h_1 + h_2$

c_9 : $\wedge\ (\neg a \vee \neg c)$

DL 1: $\boxed{a \geq 1} \xrightarrow{c_9} \boxed{c \leq 0} \xrightarrow{c_2} \boxed{b \leq 0} \xrightarrow{c_4} \boxed{x \geq -2}$

Satisfiability Modulo Theory – iSAT

$$
\begin{array}{ll}
c_1: & (\neg a \vee \neg c \vee d) \\
c_2: & \wedge\ (\neg a \vee \neg b \vee c) \\
c_3: & \wedge\ (\neg c \vee \neg d) \\
c_4: & \wedge\ (b \vee x \geq -2) \\
c_5: & \wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2) \\
 & \\
c_6: & \wedge\ h_1 = x^2 \\
c_7: & \wedge\ h_2 = -2 \cdot y \\
c_8: & \wedge\ h_3 = h_1 + h_2 \\
 & \\
c_9: & \wedge\ (\neg a \vee \neg c)
\end{array}
$$



Satisfiability Modulo Theory – iSAT

c_1 : $(\neg a \vee \neg c \vee d)$

c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$

c_3 : $\wedge\ (\neg c \vee \neg d)$

c_4 : $\wedge\ (b \vee x \geq -2)$

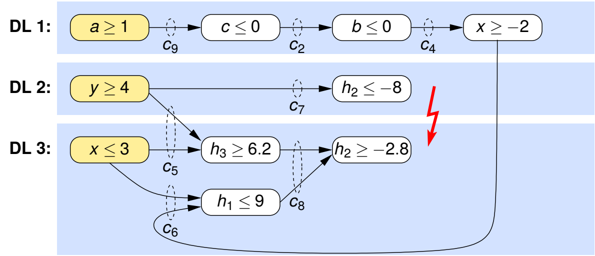
c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

c_6 : $\wedge\ h_1 = x^2$

c_7 : $\wedge\ h_2 = -2 \cdot y$

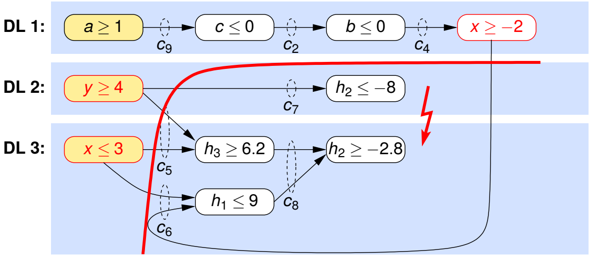
c_8 : $\wedge\ h_3 = h_1 + h_2$

c_9 : $\wedge\ (\neg a \vee \neg c)$



Satisfiability Modulo Theory – iSAT

$$
\begin{array}{rl}
c_1: & (\neg a \vee \neg c \vee d) \\
c_2: & \wedge\ (\neg a \vee \neg b \vee c) \\
c_3: & \wedge\ (\neg c \vee \neg d) \\
c_4: & \wedge\ (b \vee x \geq -2) \\
c_5: & \wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2) \\
c_6: & \wedge\ h_1 = x^2 \\
c_7: & \wedge\ h_2 = -2 \cdot y \\
c_8: & \wedge\ h_3 = h_1 + h_2 \\
c_9: & \wedge\ (\neg a \vee \neg c) \\
c_{10}: & \wedge\ (x < -2 \vee y < 4 \vee x > 3)
\end{array}
$$



$\leftarrow$ Conflict clause = symbolic description

of a rectangular region of the search space

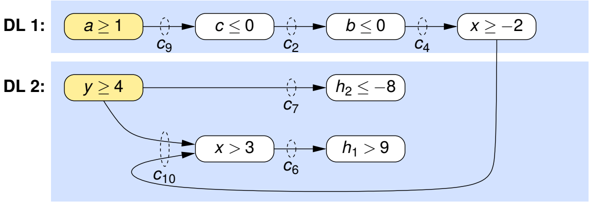
which is excluded from future search

Satisfiability Modulo Theory – iSAT

c_1 : $(\neg a \vee \neg c \vee d)$
c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$
c_3 : $\wedge\ (\neg c \vee \neg d)$
c_4 : $\wedge\ (b \vee x \geq -2)$
c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

c_6 : $\wedge\ h_1 = x^2$
c_7 : $\wedge\ h_2 = -2 \cdot y$
c_8 : $\wedge\ h_3 = h_1 + h_2$

c_9 : $\wedge\ (\neg a \vee \neg c)$
c_{10} : $\wedge\ (x < -2 \vee y < 4 \vee x > 3)$



UNI FREIBURG

Satisfiability Modulo Theory – iSAT

c_1 : $(\neg a \vee \neg c \vee d)$

c_2 : $\wedge\ (\neg a \vee \neg b \vee c)$

c_3 : $\wedge\ (\neg c \vee \neg d)$

c_4 : $\wedge\ (b \vee x \geq -2)$

c_5 : $\wedge\ (x \geq 4 \vee y \leq 0 \vee h_3 \geq 6.2)$

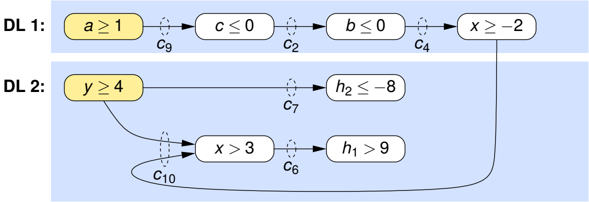
c_6 : $\wedge\ h_1 = x^2$

c_7 : $\wedge\ h_2 = -2 \cdot y$

c_8 : $\wedge\ h_3 = h_1 + h_2$

c_9 : $\wedge\ (\neg a \vee \neg c)$

c_{10} : $\wedge\ (x < -2 \vee y < 4 \vee x > 3)$



- 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

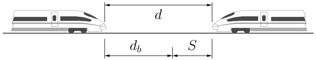
- All variables have to be bounded initially
- Reliable results due to outward rounding
- Further features
  - Clever normalization rules
  - Continue search after "unknown"
  - Proof of unsatisfiability
  - Unbounded model checking using interpolants
  - Handling of stochastic constraint systems
  - Parallelization based on message passing

UNI FREIBURG

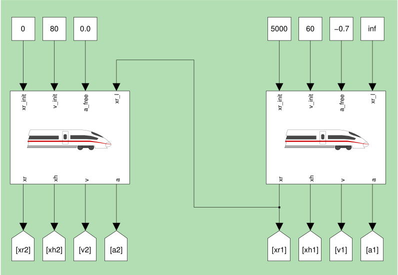
Hybrid System Verification

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

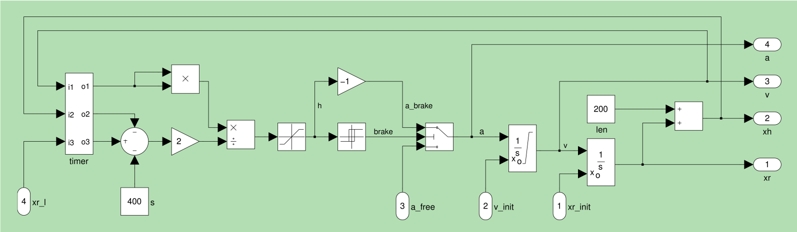


Top-level view of the Matlab/Simulink model for two trains

Hybrid System Verification

Example: Train Separation in Absolute Braking Distance

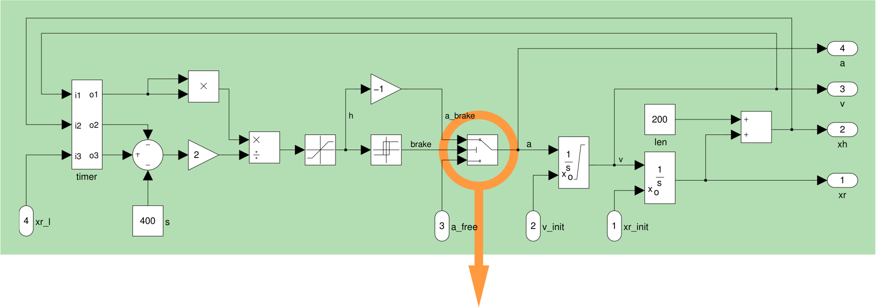
- Model of controller and train dynamics



- Safety property to be checked:
  Does the controller guarantee that collisions aren't possible?

Hybrid System Verification

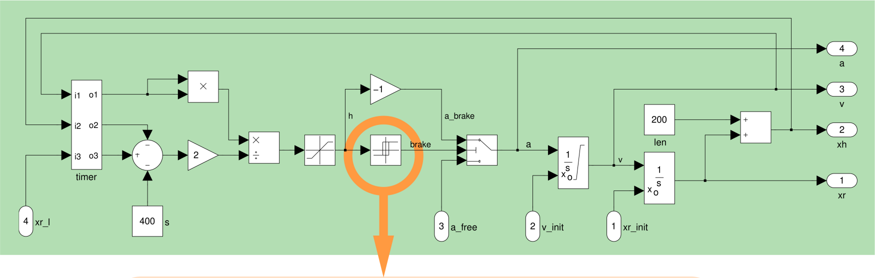
Example: Train Separation in Absolute Braking Distance




```
-- 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

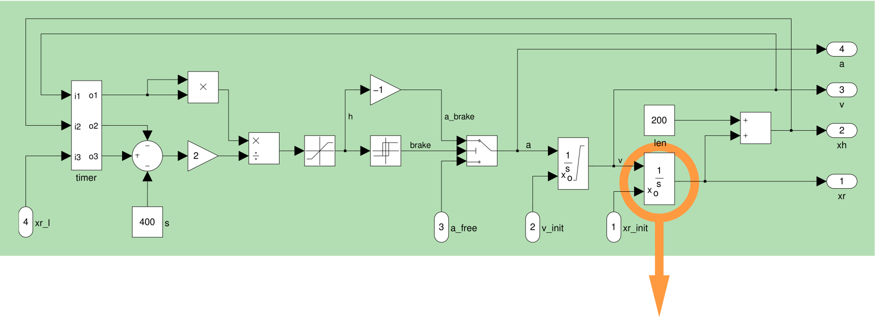
Example: Train Separation in Absolute Braking Distance




```
-- 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 >= param_on ) -> ( is_on' and  brake);
(!is_on and h <  param_on ) -> (!is_on' and !brake);
( is_on and h <= param_off) -> (!is_on' and !brake);
( is_on and h >  param_off) -> ( is_in' and  brake);
```

Hybrid System Verification

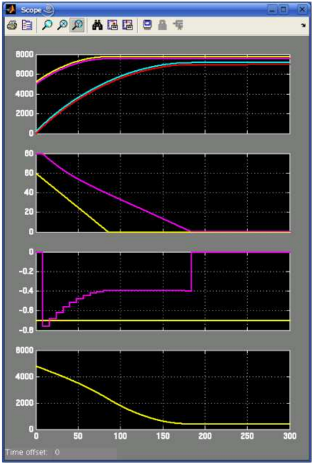
Example: Train Separation in Absolute Braking Distance



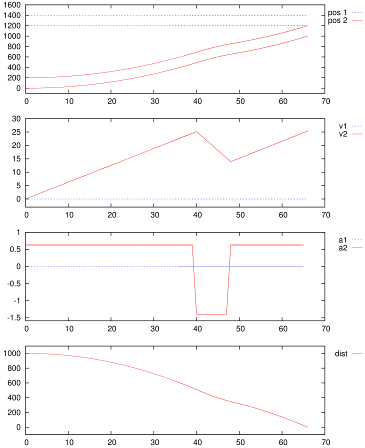

```
-- Euler approximation of integrator block

xr' = xr + dt * v;
```

Hybrid System Verification

Example: Train Separation in Absolute Braking Distance



Simulation



Error trace found by iSAT

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

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

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

Core Algorithms

MaxSAT in a Nutshell

Max-SAT

- Given a CNF φ, find a truth assignment for all variables that satisfies the maximum number of clauses within φ

Variants of Max-SAT

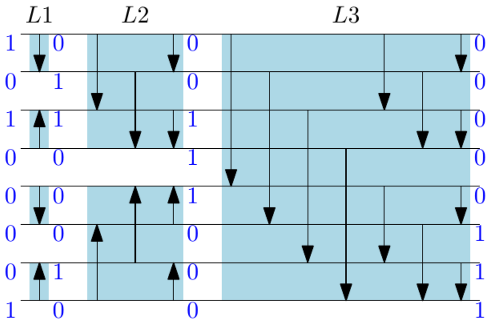
- Partial Max-SAT
  - φ consists of hard and soft clauses
  - All hard clauses must be satisfied
  - Maximize number of satisfied soft clauses
- Weighted Max-SAT
- Weighted Partial Max-SAT

MaxSAT in a Nutshell

Solving (Partial) Max-SAT using SAT Algorithms

- Each soft clause gets extended by a fresh "trigger" variable: $(x_1 \vee x_2) \rightsquigarrow (t_1 \vee x_1 \vee x_2)$
- 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

Bitonic Sorting Network



- Each arrow in the example above represents a comparator (half adder):

$$comp(x_1, x_2, y_1, y_2) \leftrightarrow ((y_1 \leftrightarrow x_1 \vee x_2) \wedge (y_2 \leftrightarrow x_1 \wedge x_2))$$

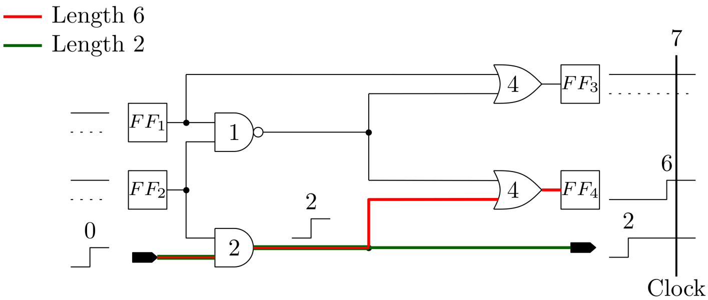
- Using Tseitin encoding each comparator can be modeled with 2 auxiliary variables & 6 clauses

Path Compaction

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

Path Compaction

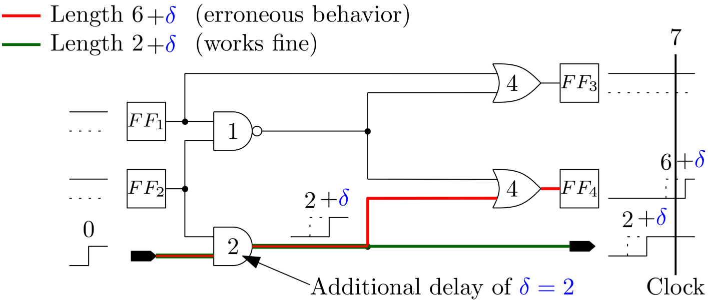
Sensitizable Paths and Small Delay Faults



- 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



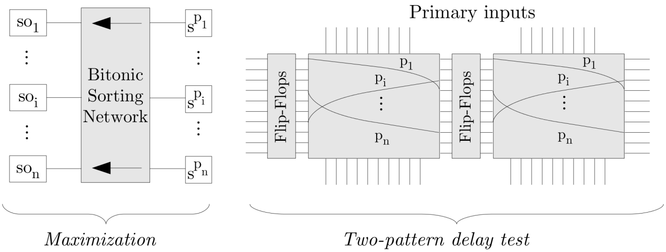
- Small delay faults: Assume additional delay for one gate
- Output transition too late for clock
- The longer the path the higher the detection quality
- Two-pattern delay test

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

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

Core Algorithms

QBF in a Nutshell

Quantified Boolean Formula (QBF)

- Extension of SAT where the variables are either universal or existential quantified
- Example
  - $\Psi = \underbrace{\exists x_1 \forall x_2, x_3 \exists x_4, \ldots, x_n}_{prefix} \underbrace{\varphi(x_1, \ldots, x_n)}_{matrix(CNF)}$
- Semantics (for this particular example)
  - Ψ is satisfied iff there exists one assignment for x_1 such that for every assignment of x_2 and x_3, there exists one assignment for $x_4, \ldots, x_n$, such that φ is satisfied

Test Pattern Relaxation using QBF

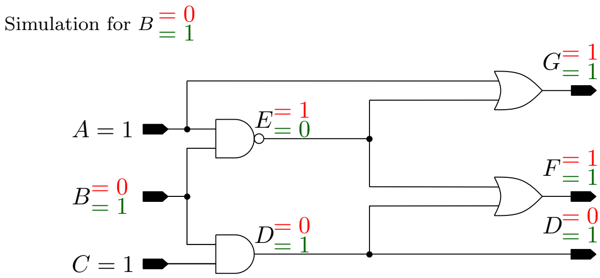
Motivation

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

Fault model considered here

- Again, small-delay Faults

Modeling Don't Cares with QBF

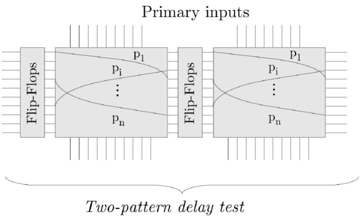


$\Rightarrow$ *F* can be set to 1, even if *B* is unspecified!

$\Rightarrow$ Don't cares can be represented by $\forall$ variables

$$\Rightarrow \underbrace{\exists\{A, C\}\forall\{B\}\exists\{D, E, F, G\}.}_{\text{Prefix}} \underbrace{\varphi(A, \ldots, G)}_{\text{Tseitin encoding}} \wedge \underbrace{(F)}_{\text{property}}$$

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 sensitizable:

  Universally quantified inputs: Excluded from test cube

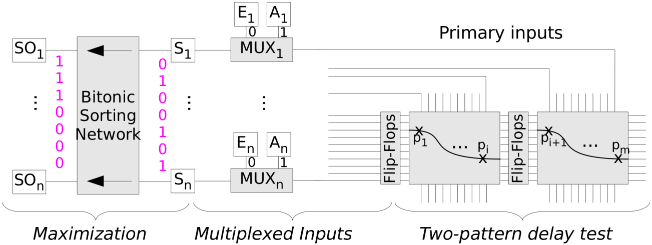
  Existential quantified inputs: Test cube
- But: The quantifier of a variable cannot be changed in QBF

⇒ Unspecified inputs are not known a-priori

⇒ Which inputs have to be quantified universally?

UNI FREIBURG

Test Pattern Relaxation using QBF



$\Psi = \exists SO_1,\ldots,SO_n,S_1,\ldots,S_n,E_1,\ldots,E_n \forall A_1,\ldots,A_n \exists \ldots \varphi_{circ.} \wedge \varphi_{prop.} \wedge \varphi_{mux} \wedge \varphi_{bsn} \wedge SO_k$

- Dynamic choice of (un-)specified inputs using multiplexers
- Select input S_i switches between specified ($S_i = 0 \rightsquigarrow \exists E_i$) and unspecified ($S_i = 1 \rightsquigarrow \forall A_i$) 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 $k+1$ ($SO_{k+1} = 0$)

$\Rightarrow$ Optimal test cube, i.e., maximum number of don't cares

Outline

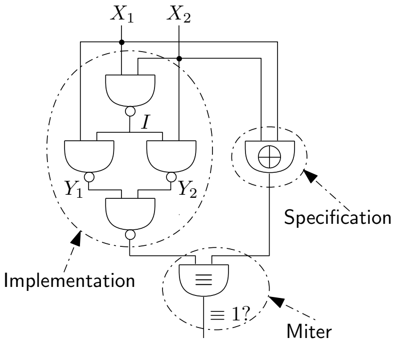
Applications

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

SAT	MaxSAT	#SAT	QBF	DQBF	SMT

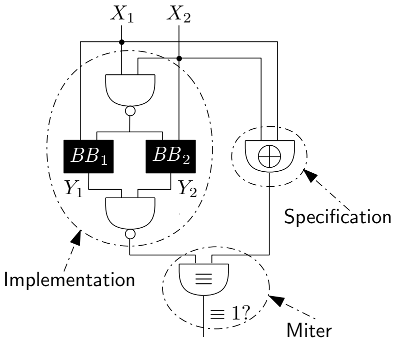
Core Algorithms

Motivation – Equivalence Checking



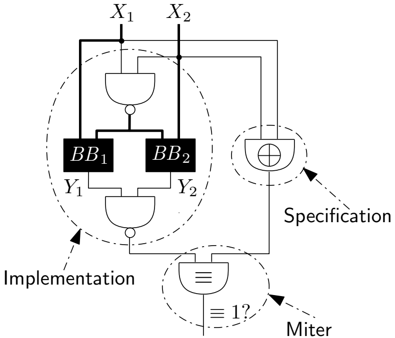
Are implementation and specification equivalent?

Motivation – Partial Equivalence Checking



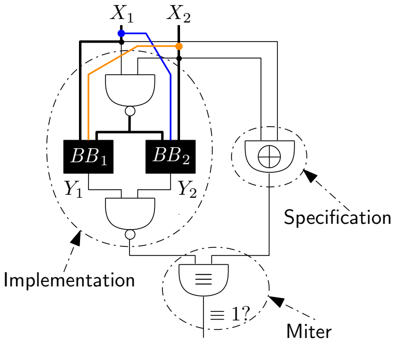
Realizability, i.e. are there implementations of the black boxes (BBs) such that implementation and specification are equivalent?

QBF vs. Dependency-QBF (DQBF)



- Expressible with QBF

QBF vs. Dependency-QBF (DQBF)

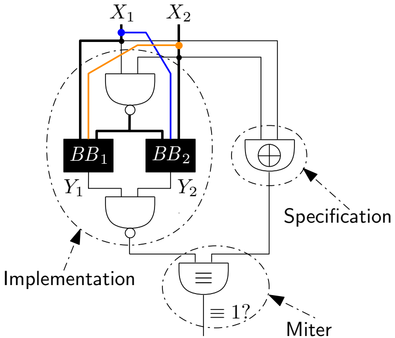


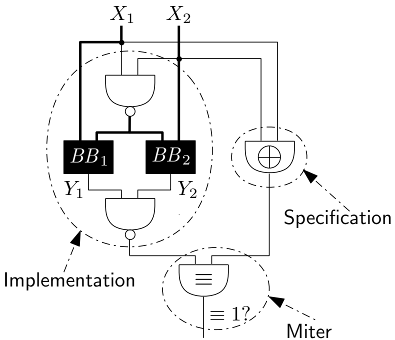
- Expressible with QBF

⇒ Approximation

- BBs read all inputs

QBF vs. Dependency-QBF (DQBF)





- Expressible with QBF

$\Rightarrow$ Approximation

- BBs read all inputs

- Expressible with DQBF

$\Rightarrow$ 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

$$\psi_{QBF} = \overbrace{\forall x_1 \forall x_2 \exists y_1 \exists y_2}^{Q} : \varphi$$

DQBF

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

$$\psi_{DQBF} = \overbrace{\forall x_1 \forall x_2 \exists y_1 \underbrace{\{x_1\}} \exists y_2 \underbrace{\{x_2\}}}^{Q} : \varphi$$

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 $u \in dep(e)$ the assignment of the $\exists$ variable e has to be the same
2) For different assignments of at least one $\forall$ variable $u \in dep(e)$ 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 $\not\Rightarrow$ realizability

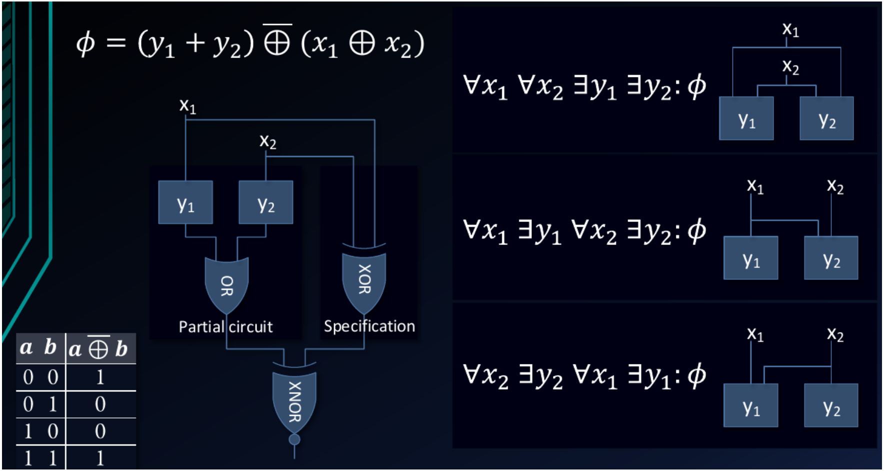
DQBF

- BBs read only affecting signals

- UNSAT $\Rightarrow$ unrealizability
- SAT $\Rightarrow$ realizability

For one black box QBF is as accurate as DQBF!

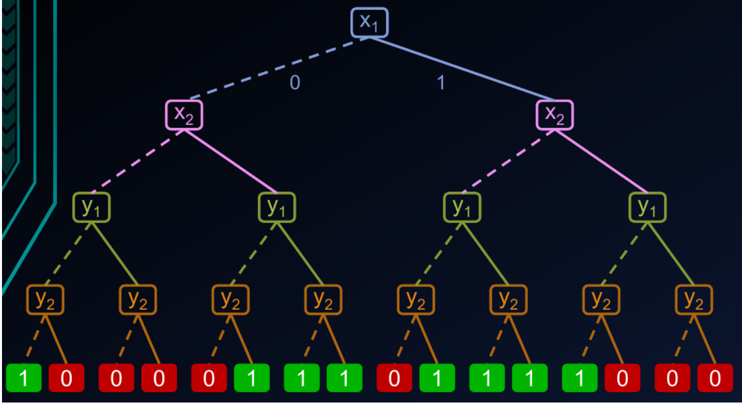
DQBF-based Partial Equiv. Checking – Example



UNI FREIBURG

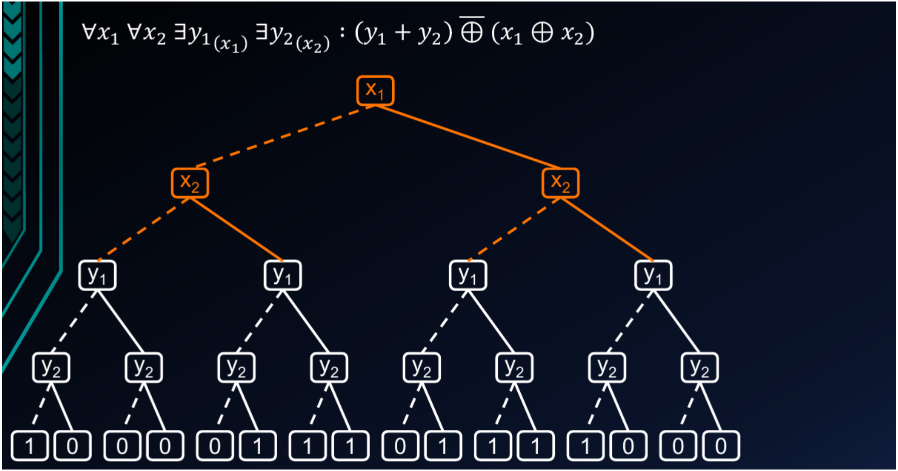
DQBF-based Partial Equiv. Checking – Example

$$\forall x_1\ \forall x_2\ \exists y_1\ \exists y_2 : (y_1 + y_2)\ \overline{\oplus}\ (x_1 \oplus x_2)$$

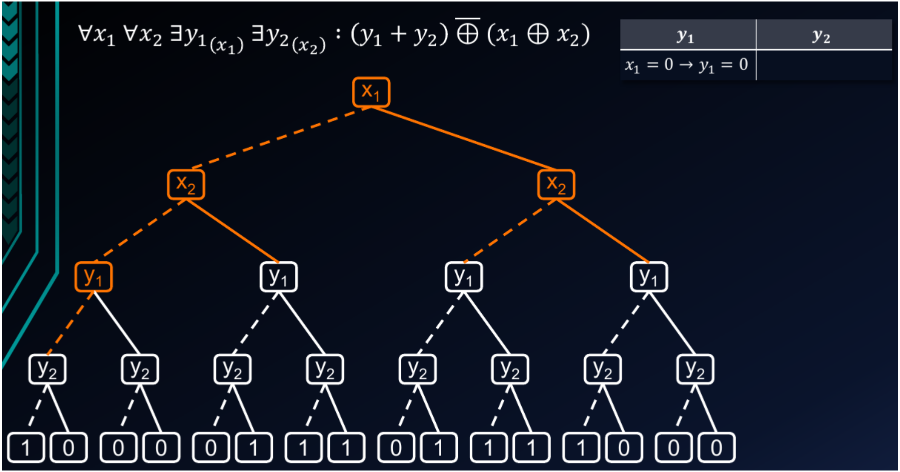


x_1	x_2	y_1	y_2	ϕ
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

DQBF-based Partial Equiv. Checking – Example



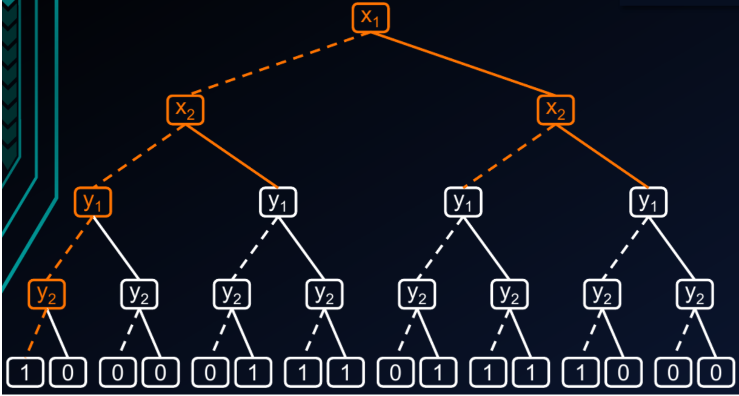
DQBF-based Partial Equiv. Checking – Example



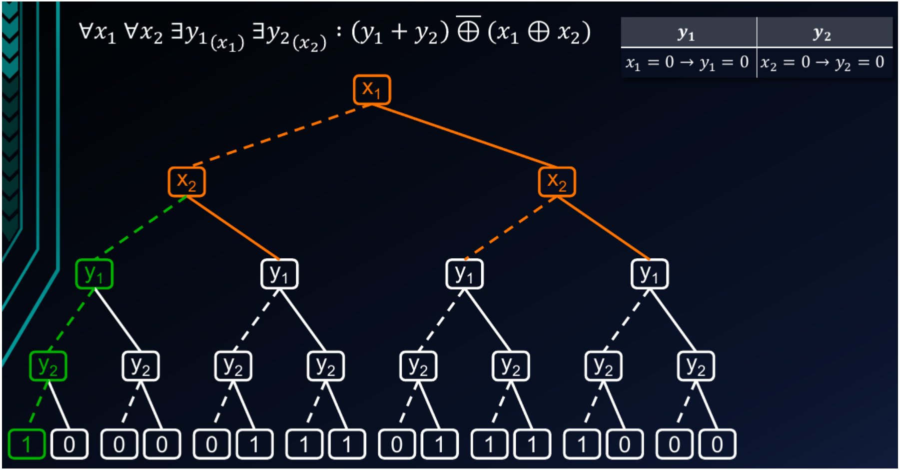
DQBF-based Partial Equiv. Checking – Example

$$\forall x_1\, \forall x_2\, \exists y_{1(x_1)}\, \exists y_{2(x_2)} : (y_1 + y_2)\, \overline{\oplus}\, (x_1 \oplus x_2)$$

y_1	y_2
$x_1 = 0 \rightarrow y_1 = 0$	$x_2 = 0 \rightarrow y_2 = 0$



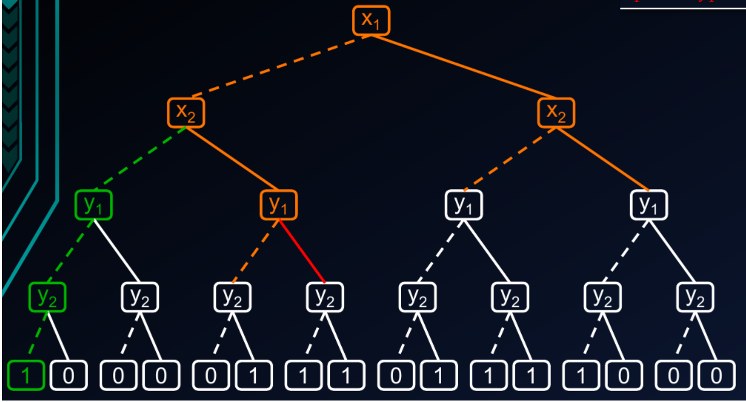
DQBF-based Partial Equiv. Checking – Example



DQBF-based Partial Equiv. Checking – Example

$\forall x_1 \, \forall x_2 \, \exists y_{1(x_1)} \, \exists y_{2(x_2)} : (y_1 + y_2) \, \overline{\oplus} \, (x_1 \oplus x_2)$

y_1	y_2
$x_1 = 0 \rightarrow y_1 = 0$	$x_2 = 0 \rightarrow y_2 = 0$

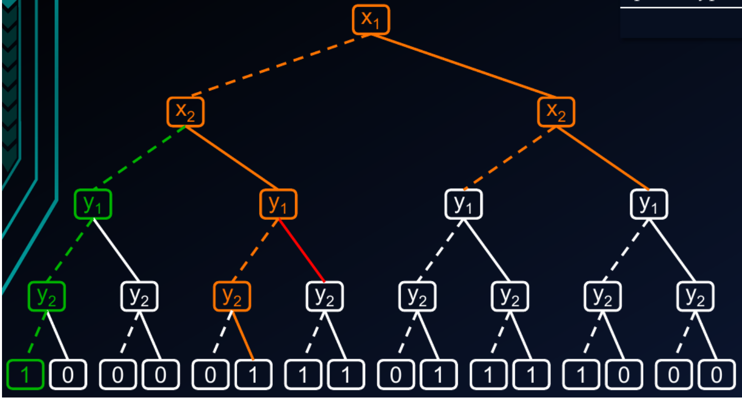


UNI FREIBURG

DQBF-based Partial Equiv. Checking – Example

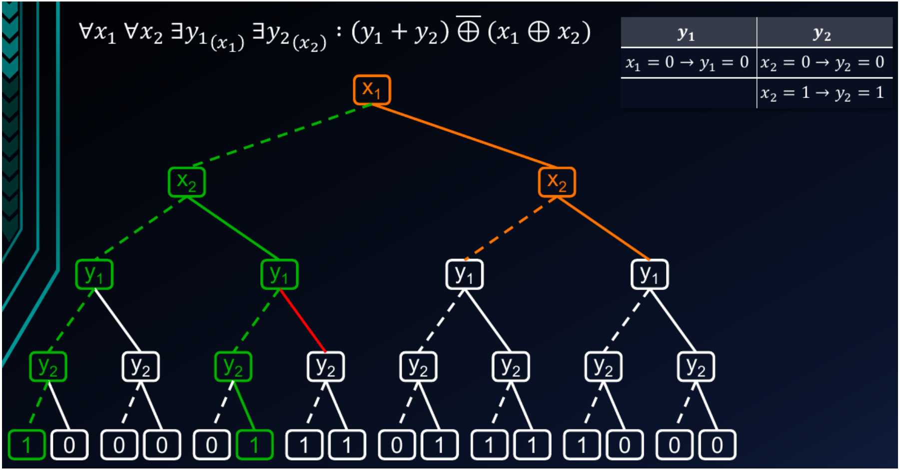
$$\forall x_1\, \forall x_2\, \exists y_{1(x_1)}\, \exists y_{2(x_2)} : (y_1 + y_2)\, \overline{\oplus}\, (x_1 \oplus x_2)$$

y_1	y_2
$x_1 = 0 \rightarrow y_1 = 0$	$x_2 = 0 \rightarrow y_2 = 0$
	$x_2 = 1 \rightarrow y_2 = 1$



UNI FREIBURG

DQBF-based Partial Equiv. Checking – Example

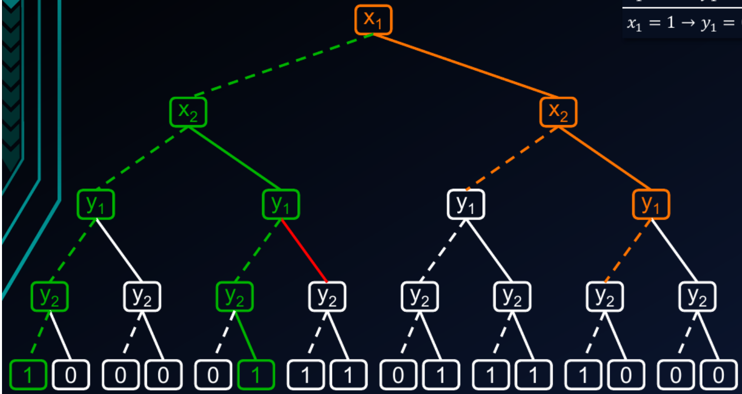


UNI FREIBURG

DQBF-based Partial Equiv. Checking – Example

$$\forall x_1\, \forall x_2\, \exists y_{1(x_1)}\, \exists y_{2(x_2)} : (y_1 + y_2)\, \overline{\oplus}\, (x_1 \oplus x_2)$$

y_1	y_2
$x_1 = 0 \rightarrow y_1 = 0$	$x_2 = 0 \rightarrow y_2 = 0$
$x_1 = 1 \rightarrow y_1 = 0$	$x_2 = 1 \rightarrow y_2 = 1$

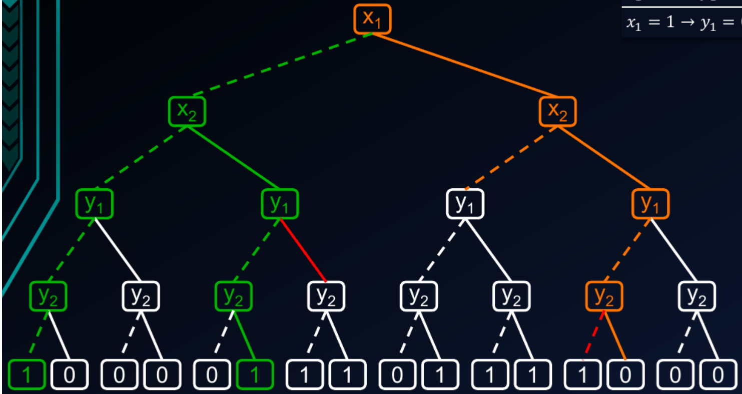


UNI FREIBURG

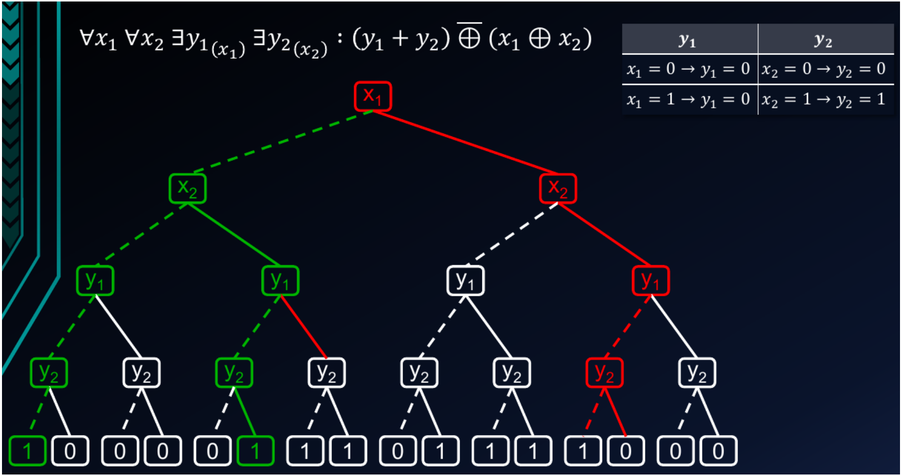
DQBF-based Partial Equiv. Checking – Example

$$\forall x_1\ \forall x_2\ \exists y_{1(x_1)}\ \exists y_{2(x_2)} : (y_1 + y_2)\ \overline{\oplus}\ (x_1 \oplus x_2)$$

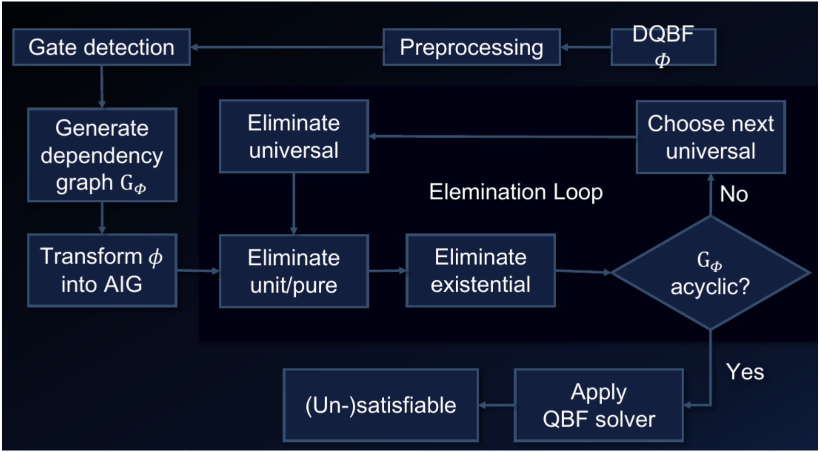
y_1	y_2
$x_1 = 0 \rightarrow y_1 = 0$	$x_2 = 0 \rightarrow y_2 = 0$
$x_1 = 1 \rightarrow y_1 = 0$	$x_2 = 1 \rightarrow y_2 = 1$



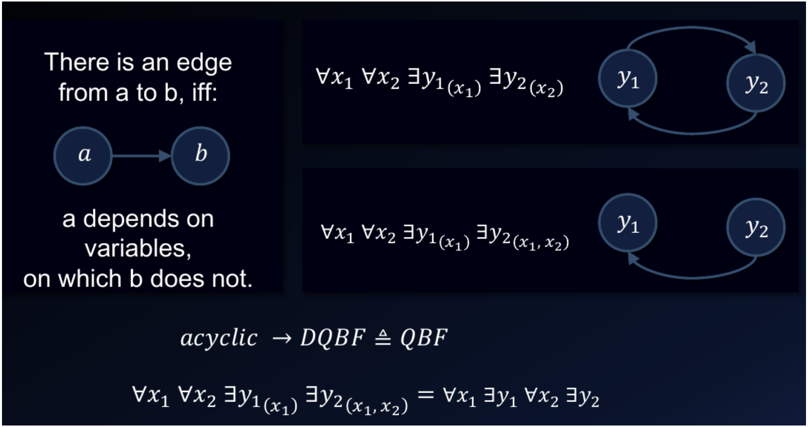
DQBF-based Partial Equiv. Checking – Example



Henkin Quantified Solver (HQS)



Main Idea behind HQS – Acyclic Dependency Graph



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

SAT	MaxSAT	#SAT	QBF	DQBF	SMT

Core Algorithms

#SAT in a Nutshell

#SAT

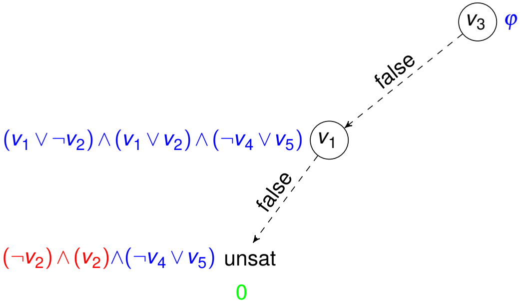
- Given a CNF φ, count how many disjoint truth assignments satisfy φ
- #SAT solver have to continue search after one solution has been found
- With n variables, φ can have up to 2^n satisfying assignments
- #SAT corresponds to model counting, not enumerating all satisfying assignments
- Accelerating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

#SAT – Example

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$

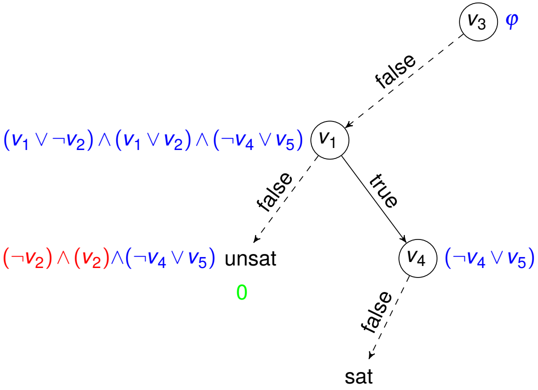
#SAT – Example

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$



#SAT – Example

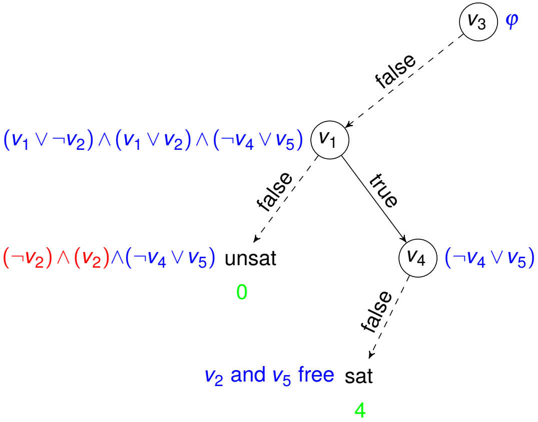
$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$



UNI FREIBURG

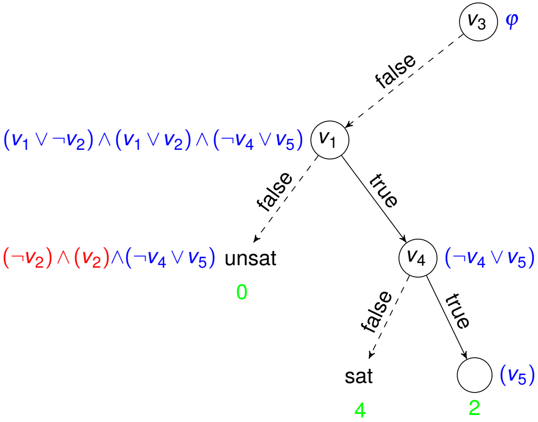
#SAT – Example

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$



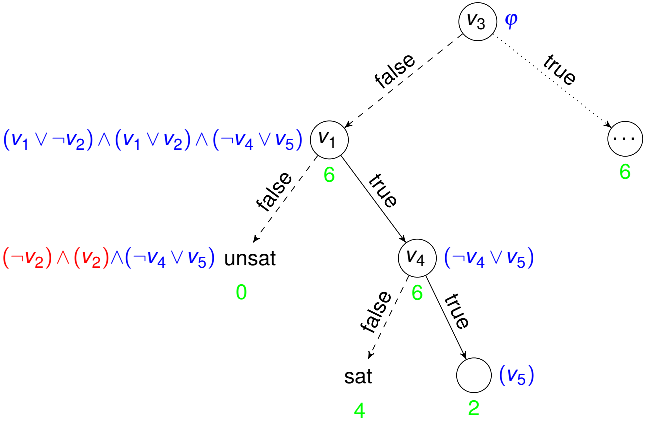
#SAT – Example

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$



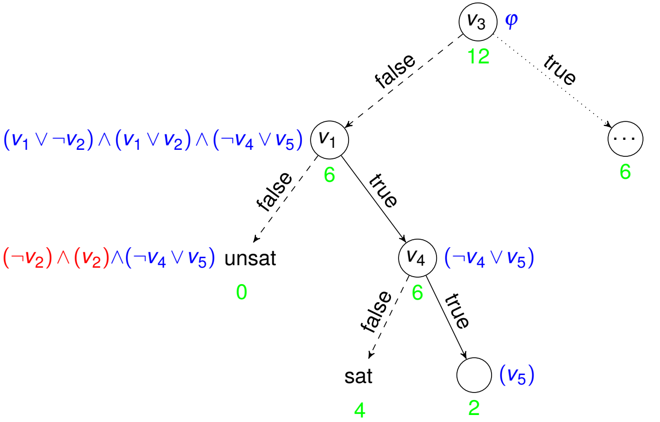
#SAT – Example

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



#SAT – Example

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_1 \vee v_2 \vee v_3) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_3 \vee v_5)$$



- $mc(\varphi) = 12$

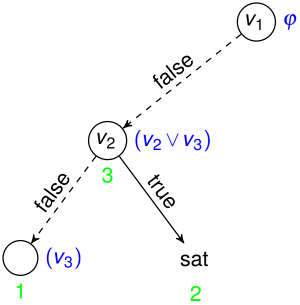
#SAT – Caching

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

UNI FREIBURG

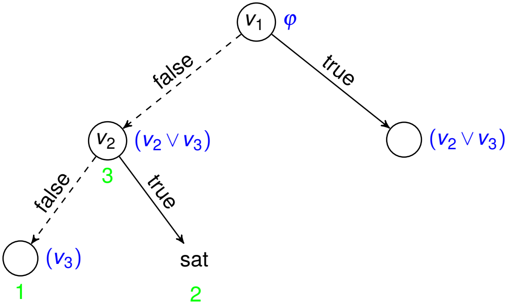
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- $\varphi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$



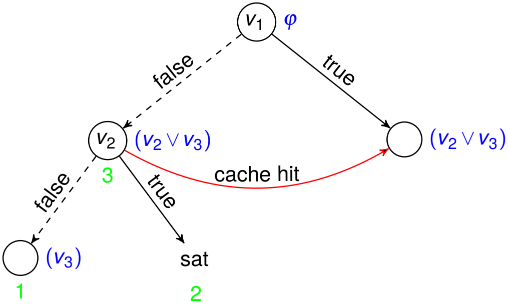
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- $\varphi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$



#SAT – Caching

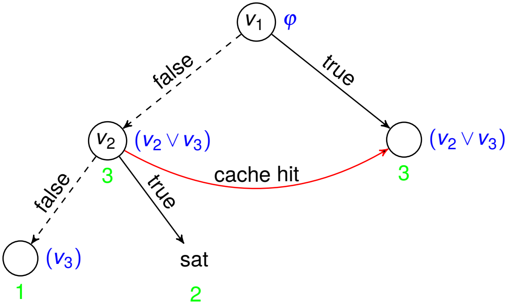
- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- $\varphi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$



UNI FREIBURG

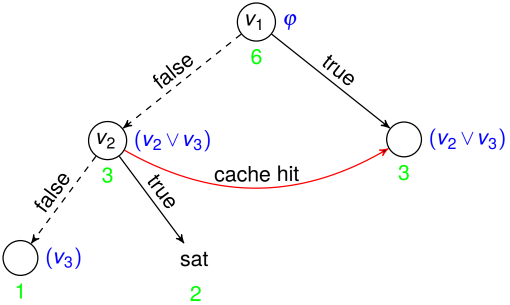
#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- $\varphi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$



#SAT – Caching

- Store model counts of sub-formulas in a cache
- Do not compute the result for the same sub-formula twice
- $\varphi = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$



#SAT – Component Analysis

- The formula might split into disjoint sub-formulas

#SAT – Component Analysis

- The formula might split into disjoint sub-formulas
  - $\varphi = (\neg p_2 \vee a_2) \wedge (a_1 \vee a_2 \vee a_3) \wedge (b_1) \wedge (\neg b_3 \vee b_4) \wedge (p_2 \vee \neg b_2)$

#SAT – Component Analysis

- The formula might split into disjoint sub-formulas
  - $\varphi = (\neg p_2 \vee a_2) \wedge (a_1 \vee a_2 \vee a_3) \wedge (b_1) \wedge (\neg b_3 \vee b_4) \wedge (p_2 \vee \neg b_2)$
  - Assignment: $p_2 = \mathit{false}$

#SAT – Component Analysis

- The formula might split into disjoint sub-formulas
  - $\varphi = (\neg p_2 \vee a_2) \wedge (a_1 \vee a_2 \vee a_3) \wedge (b_1) \wedge (\neg b_3 \vee b_4) \wedge (p_2 \vee \neg b_2)$
  - Assignment: $p_2 = \mathit{false}$
  - Sub-formulas:

    $\varphi_1 = (a_1 \vee a_2 \vee a_3)$
    $\varphi_2 = (b_1) \wedge (\neg b_3 \vee b_4) \wedge (\neg b_2)$

#SAT – Component Analysis

- The formula might split into disjoint sub-formulas
  - $\varphi = (\neg p_2 \vee a_2) \wedge (a_1 \vee a_2 \vee a_3) \wedge (b_1) \wedge (\neg b_3 \vee b_4) \wedge (p_2 \vee \neg b_2)$
  - Assignment: $p_2 = \mathit{false}$
  - Sub-formulas:

    $\varphi_1 = (a_1 \vee a_2 \vee a_3)$
    $\varphi_2 = (b_1) \wedge (\neg b_3 \vee b_4) \wedge (\neg b_2)$
- Model count is computed by multiplying results for sub-formulas:

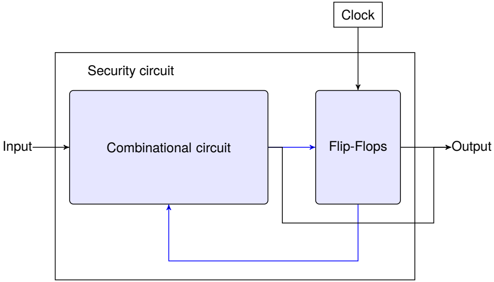
  $mc(\varphi|_{p_2=\mathit{false}}) = mc(\varphi_1) \cdot mc(\varphi_2) = 7 \cdot 3 = 21$

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

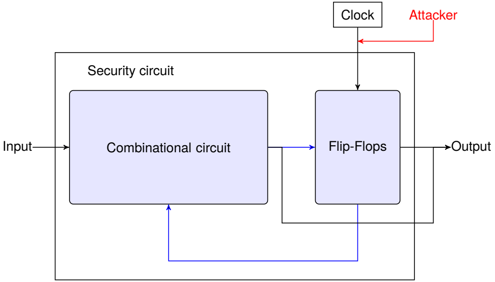
- Extract secret information from a security circuit (AES, . . .)
- Inject fault by increasing the clock frequency
- Incorrect output allows for calculation of secret



UNI FREIBURG

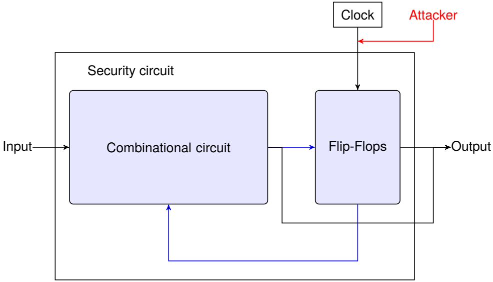
Security Issues – Fault Injection

- 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

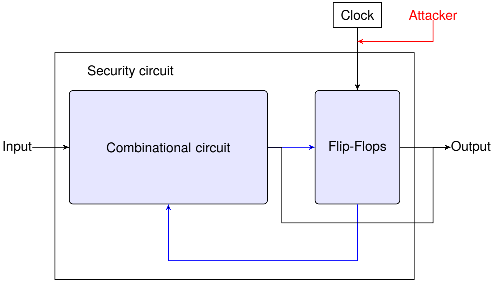


- Flip-flops store value on rising clock edge

UNI FREIBURG

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?

UNI FREIBURG

Security Issues – Fault Injection

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

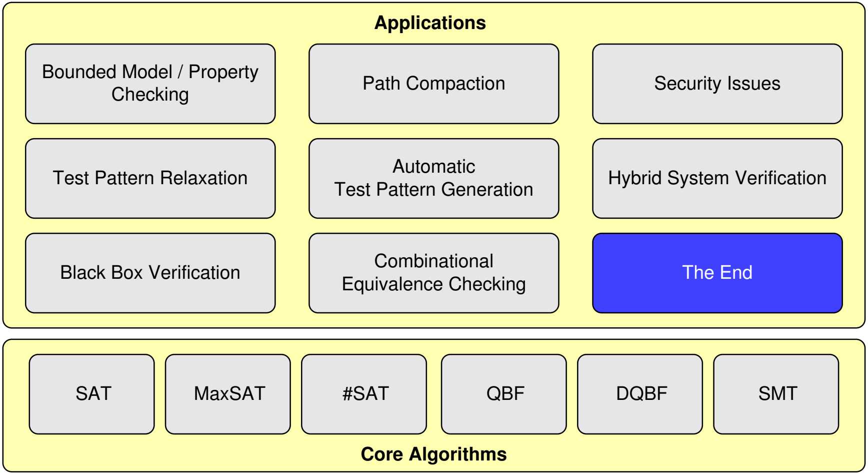
[1] M. Sauer et al. "Small-Delay-Fault ATPG with Waveform Accuracy". In: ICCAD 2012.

Security Issues – Fault Injection

1. Encode combinational circuit and its timing as CNF formula φ with the tool WaveSAT[1]
2. Make φ satisfiable iff at least one fault is injected
3. Add conditions for outputs that must be correct
4. Calculate number of satisfying assignments $mc(\varphi)$
5. $P(\mathit{Successful\ Injection}) = \dfrac{mc(\varphi)}{2^{\#\text{circuit inputs}}}$

[1] M. Sauer et al. "Small-Delay-Fault ATPG with Waveform Accuracy". In: ICCAD 2012.

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
SAT
MaxSAT
#SAT
QBF
DQBF
SMT
Core Algorithms

Some Papers…

[Abraham, Schubert, Becker, Fränzle, Herde. *Parallel SAT Solving in BMC*. Logic & Computation, 2011]

[Burchard, Schubert, Becker. *Laissez-Faire Caching for Parallel #SAT Solving*. SAT, 2015]

[Feiten, Sauer, Schubert, Czutro, Boehl, Polian, Becker. *#SAT-Based Vulnerability Analysis of Security Components – A Case Study*. IEEE DFTS, 2012]

[Fränzle, Herde, Teige, Ratschan, Schubert. *Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure*. JSAT, 2007]

[Gitina, Wimmer, Reimer, Sauer, Scholl, Becker. *Solving DQBF Through Quantifier Elimination*. DATE, 2015]

[Kalinnik, Schubert, Abraham, Wimmer, Becker. *Picoso - A Parallel Interval Constraint Solver*. PDPTA, 2009]

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

[Lewis, Schubert, Becker. *Multithreaded SAT Solving*. ASP-DAC, 2007]

[Reimer, Sauer, Schubert, Becker. *Incremental Encoding and Solving of Cardinality Constraints*. ATVA, 2014]

[Reimer, Sauer, Schubert, Becker. *Using MaxBMC for Pareto-Optimal Circuit Initialization*. DATE, 2014]

[Sauer, Czutro, Schubert, Hillebrecht, Polian, Becker. *SAT-based Analysis of Sensitisable Paths*. IEEE Design & Test of Computers, 2013]

[Sauer, Reimer, Schubert, Polian, Becker. *Efficient SAT-Based Dynamic Compaction and Relaxation for Longest Sensitizable Paths*. DATE, 2103]

[Sauer, Reimer, Polian, Schubert, Becker. *Provably Optimal Test Cube Generation Using Quantified Boolean Formula Solving*. ASP-DAC, 2013]

[Schubert, Lewis, Becker. *Parallel SAT Solving with Threads and Message Passing*. JSAT, 2009]