

Introduction to SAT and SMT Solvers

Interfacing Yosys and SMT Solvers
for BMC and more using SMT-LIB 2.5

Clifford Wolf

Abstract

Bounded model checking and other SAT-based formal methods are an important part of today's digital design methodologies. SMT solvers add theories – such as bit-vector arithmetic and arrays – to the basic functionality of SAT solvers. This allows for encoding problems in a more efficient way and enables solvers to better understand the structure of a problem. SMT-LIB (currently in version 2.5) is a language for SMT problems. It is supported by virtually all SMT solvers, because it is used in the annually SMT competition where the performance of different SMT solvers for the same suite of SMT-LIB benchmarks is compared.

Most applications using an SMT solver bind to one specific solver using its C/C++ API. Yosys on the other hand comes with a methodology for using SMT-LIB as interface to the SMT solver, thus enabling flows with interchangeable SMT solvers, avoiding lock-in on individual solver packages and allowing users to choose the solver that performs best for the type of problem at hand.

This presentation covers a quick introduction to SAT and SMT solving, the SMT-LIB language and how to use it to interact with an SMT solver, how to generate SMT-LIB code from Verilog HDL using Yosys, and how to create complex proofs using simple Python scripts that control any SMT solver that understands the SMT-LIB language.

About the Presenter: Clifford Wolf is the author of Yosys (a framework for Verilog HDL synthesis and more) and Project IceStorm (reverse engineered bit-stream documentation for Lattice iCE40 FPGAs), among other open source projects. He is probably best known for being the original author of OpenSCAD. In his professional life he spends his time doing mathematical modeling and writing computational FPGA cores for LIDAR devices. For this work he is also using Yosys combined with SMT solvers for verification purposes.

Overview / Outline

- Quick Introduction to Complexity Theory
 - Classes of decision problems
 - FSMs and Turing machines
 - SAT and QBF problems
- Universal solvers for hard problems
 - SAT solver and CNF representation
 - SMT solver and SMT-LIB 2.5
- Formal verification methods for digital circuits
 - Bounded model checking and temporal induction
 - Formal Verification with Yosys
 - Yosys-SMTBMC

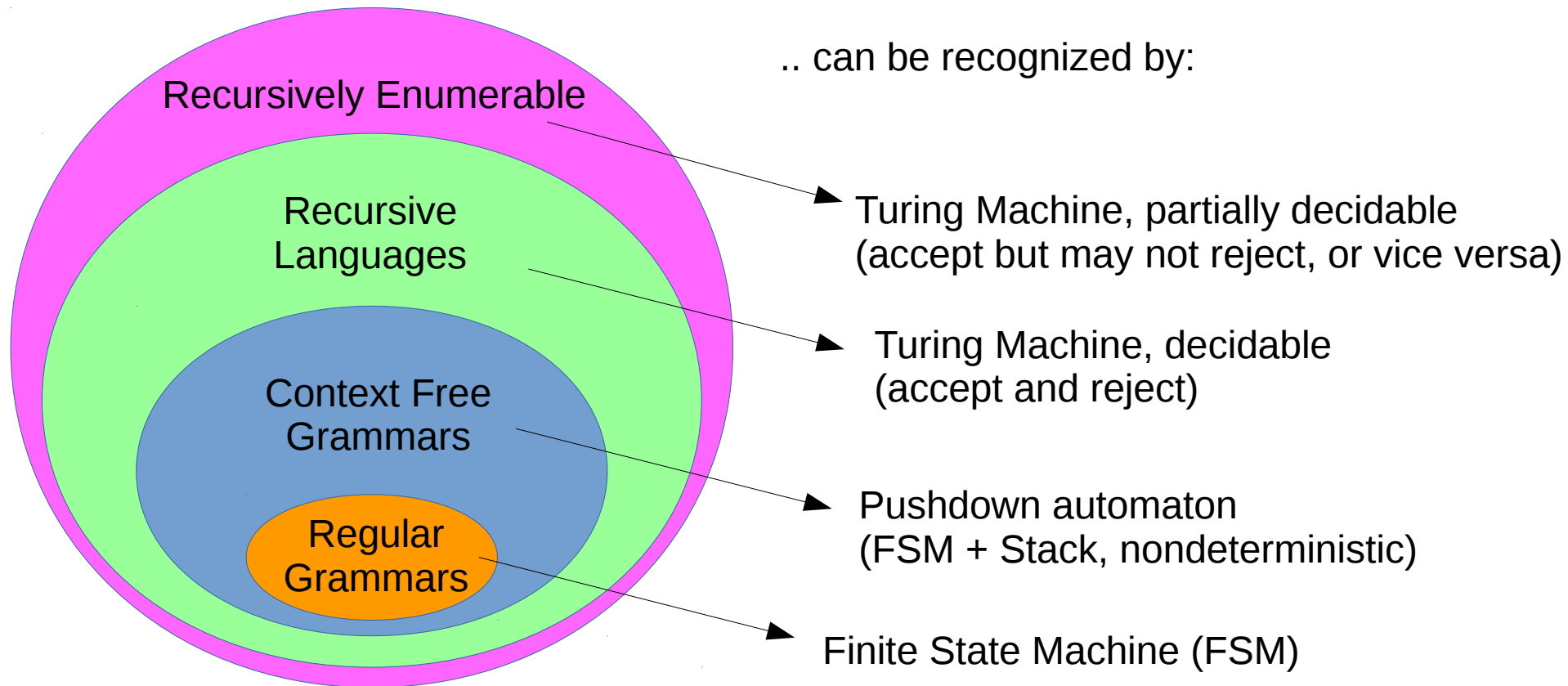
Complexity Theory

- Complexity theory is the study of computational problems and their complexity in time and space
 - Most of complexity theory is related to the classification of
 - Problems and algorithms
 - Computing machines
 - Complexity is usually expressed using Landau “Big O” notation:
 - $O(f(n))$ means that $f(n)$ is a low upper bound for the asymptotic growth rate of the problem complexity with respect to the input size n .
 - In complexity theory, often the $O(...)$ is replaced with the name of the resource, prefixed with a short identifier for the machine model. E.g. $NSPACE(f(n))$ means $f(n)$ is a low upper bound for the asymptotic growth rate of the problem complexity for a non-deterministic Turing machine.
- Landau “Big O” examples:
 - $O(n) = O(2n) = O(n + 3)$
 - $O(12n^3 + n^2 + 15n) = O(n^3)$

Decision Problems

- We are only discussing decision problems today
- Examples:
 - “Does $13 + x = 18$ have a solution?”
 - What about $x^2 = 25$? What about $x+1 = x$?
- When we also ask for the assignment of x , then this is called a *Function Problem*.
- In many cases, deciding that an assignment for x exists is similar in complexity to finding the assignment.
- Solvers like SAT and SMT solvers are defined as solving the decision problem, but provide interfaces that allow for inspection of the model, that is finding the assignment that satisfies the given set of constraints when the constraints are found to be satisfiable.

The Relationship between Formal Languages and Computability



All decision problems can be formulated as the problem of recognizing members of a formal language

Finite State Machines

- A finite set of states
 - Some of them are accepting states
- A finite set of input symbols
- A state transition table that maps
 - The current state and input symbol
 - To the next state
- The FSM accepts the input if the last state (after the entire input has been read) is an accepting state.

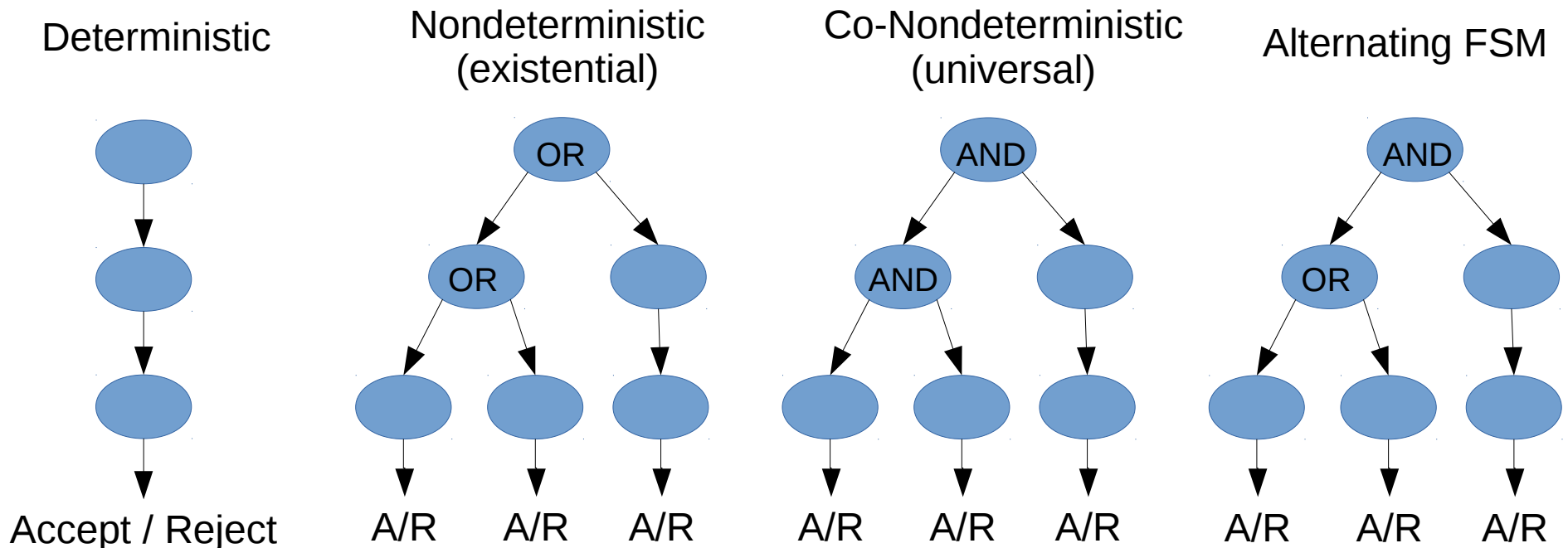
Nondeterministic FSMs

- A deterministic FSM has exactly one entry in the state transition table for each current state + current input symbol combination
- A nondeterministic FSM can have any number of transitions for each current state + current input symbol combination
- A nondeterministic FSM may also have ϵ -moves, transitions that are always taken without consuming an input symbol. (Sometimes this is classified as a separate category of FSMs.)
- Instead of a “current state” a nondeterministic FSM has a set of current states.
- A non-deterministic FSM accepts the input if any accepting state is in the set of current states after the last input symbol has been processed.

In my experience the term “nondeterministic” for this kind of machines can be confusing, as it can be understood as suggesting guesswork and/or random events being involved. Note that neither is the case!

Existential and Universal Nondeterminism, Alternating FSMs

- Usually a nondeterministic FSM is existential:
 - It accepts its input if one current state is accepting
- Also a thing: co-nondeterministic FSMs:
 - Accept input only if all current states are accepting
- Alternating FSMs:
 - Whether the reduction is existential or universal is a state property



Complexity of FSMs

- All FSMs process an input of n symbols in n steps
 - Time complexity: $O(n)$
- An FSM only needs to store the current state. The number of bits needed for that is independent of the length of the input
 - Space complexity: $O(1)$
- Deterministic FSMs, Nondeterministic FSMs, Co-Nondeterministic FSMs, and Alternating FSMs have the same computational power!
- When converting Nondeterministic FSMs to Deterministic FSMs, the state space can grow exponentially!

Turing Machines

- A Turing Machine is an FSM plus a read/write array and a data pointer
 - The array is usually called tape
 - The data pointer can be incremented and decremented
- The transition table for a Turing Machine maps
 - From current state and symbol on the tape
 - To next state, tape write operation, and tape movement
- A Turing Machine has accepting and rejecting states
- A Turing Machine halts when it reaches an accepting or rejecting state
- The Turing Machine starts with the input on the tape
- Popular equivalent extensions and variations:
 - More than one tape (for example three tapes: input, temp storage, and output)
 - Instead of tape: two or more stacks

Nondeterministic Turing Machines

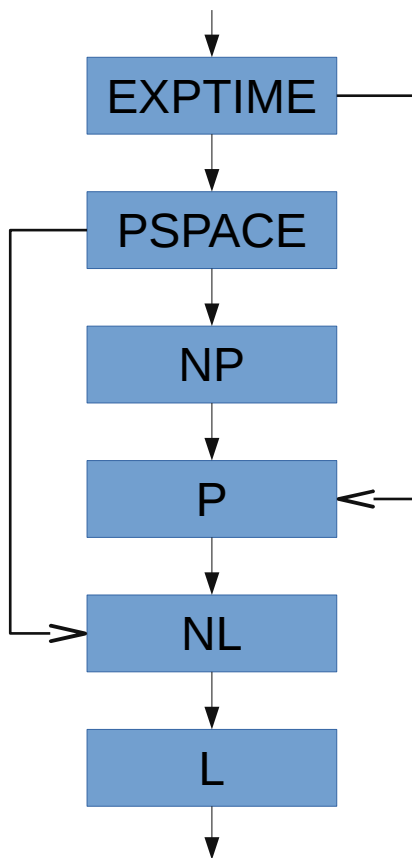
- Like FSMs, Turing Machines can be..
 - Deterministic
 - Nondeterministic
 - Co-Nondeterministic
 - Alternating
- We cannot build nondeterministic Turing Machines!
- But they are a very helpful computational model for categorizing problems, as we will see..

Note: Nondeterministic Turing Machines and Quantum Computers are different computational models that are assumed to be not equivalent!

Universal Turing Machines

- A Universal Turing Machine is a Turing Machine that can simulate
 - Any Turing Machine
 - On any Input
- Time complexity of emulating one Turing Machine on another:
 - $O(n) \rightarrow O(n \log n)$
- A Turing Machine over the alphabet $\{0, 1\}$ can emulate any Turing Machine with a finite alphabet
- Small Universal Turing Machines:
 - 2 states and 3 symbols is proven to be sufficient

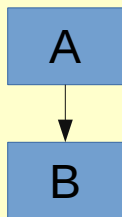
Complexity of Turing Machines



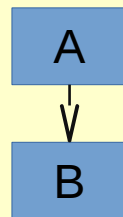
- Definitions for this complexity classes using “Big-O” space/time resource usage on deterministic or non-deterministic Turing machines:

- EXPTIME: $\text{DTIME}(2^{p(n)})$
- PSPACE: $\text{DSPACE}(p(n))$
- NP: $\text{NTIME}(p(n))$
- P: $\text{DTIME}(p(n))$
- NL: $\text{NSPACE}(\log n)$
- L: $\text{DSPACE}(\log n)$

- For most of the inclusions it is unknown if they are proper! Most famously: The “P vs NP Problem”



A is a superset of B



A is a proper superset of B

P, NP, Co-NP, PSPACE

- P
 - Polynomial time on Deterministic Turing Machine: $\text{DTIME}(p(n))$
- NP
 - Polynomial time on Nondeterministic Turing Machine: $\text{NTIME}(p(n))$
- Co-NP
 - Polynomial time on Co-Nondeterministic Turing Machine
- PSPACE
 - Polynomial time on Alternating Turing Machine: $\text{ATIME}(p(n))$
 - Polynomial space on Deterministic Turing Machine: $\text{DSpace}(p(n))$

Can we analyze FSMs?

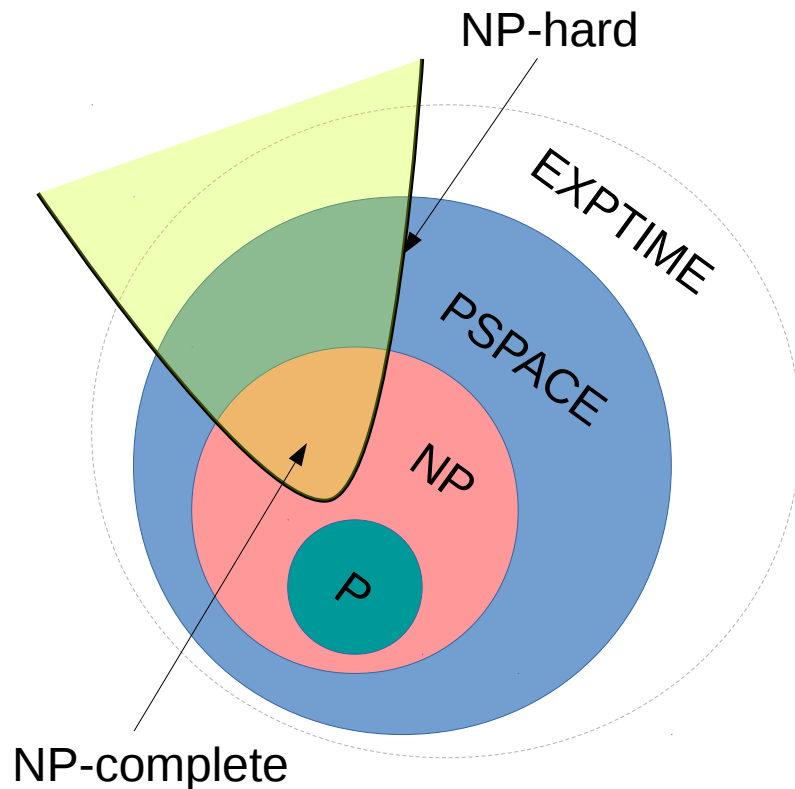
- FSMs are closed under:
 - Compliment, Union, Intersection, Concatenation, Iteration, ...
 - ... short: FSMs are closed under almost everything!
- This makes it very easy to formally analyze FSMs!
- Example: Does FSM **B** accept everything FSM **A** does accept?
 - (1) Create and simplify FSM: $(\text{not } (A \rightarrow B))$ (equal to $(A \text{ and } (\text{not } B))$)
 - (2) Does the new FSM accept anything?
 - If YES: B does not accept everything that A accepts
Everything the new FSM accepts is a counter example
It is possible to convert it into a grammar that produces counter examples
 - If NO: B does accept everything that A accepts

Can we analyze Turing Machines?

- We do not know if $P \neq NP \neq \text{Co-NP} \neq \text{PSPACE}$
 - But for practical purposes we can assume they are..
- For many, many problems we do not know it's lowest bound for time and/or space complexity!
 - But sometimes we can prove by reduction that their complexity depends on the $P \neq NP \neq \text{Co-NP} \neq \text{PSPACE}$ problem.
- For the general case it's undecidable if a Turing Machine..
 - halts for a given input or
 - is equivalent to another Turing Machine
- Or to make it short: **All non-trivial questions about Turing Machines are undecidable!** (Rice's Theorem)
 - We can not even decide if two Context Free Grammars are equivalent!

NP-hard, NP-complete

- NP-hard problem:
 - All problems in NP can be reduced to this problem in polynomial time.
- NP-complete problem:
 - NP-hard and in NP
- Similar definitions for
 - P-complete
 - PSPACE-complete
 - etc.



Complexity Quiz:

What is the complexity of the following problems?

- Minimize a deterministic FSM
- Minimize a non-deterministic FSM
- Solving Generalized Chess
- Deciding if two regular expressions are equivalent
 - Four regex operators: (ab) , $(a|b)$, (a^*) , $(a\{2\})$
- Solving the boolean satisfiability (SAT) problem
- Solving the boolean quantified formula problem
- LZW compression algorithm
 - given strings s and t , will compressing s with an LZ78 method add t to the dictionary?
- Testing if a given integer is prime
- For a given finite set of numbers: Exists a subset so that the sum is in a given interval?
- Sorting by comparing elements
- Building a binary heap
- Linear Programming
- Matrix-Matrix-Multiply

Answers after
two more slides !

The SAT Problem

- Given a boolean expression in any number of variables, is there an assignment for the variables so that the expression evaluates to true?
- Usually the expression is specified as CNF (Commutative Normal Form, in this Context also Clause Normal Form), for example:
 $(a + b + c) * (a + \text{NOT}(e)) * (e + \text{NOT}(c)) * \dots$
- Conversion of any boolean expression to an equisatisfiable CNF is linear. (Conversion to an equivalent CNF would be exponential!)
- The 3-SAT problem (max 3 literals per clause) is equivalent to the general SAT problem.
- Nomenclature:
 - Clause – A disjunction, such as $(a + b + c)$
 - Positive Literal – Non-inverted variable, such as a
 - Negative Literal – Inverted variable, such as $\text{NOT}(e)$

The QBF Problem

- The Quantified Boolean Formula (QBF) Problem is an extension to the SAT problem:
- SAT checks only for existence of a single set of assignments that satisfy the formula
- QBF can check any alternating pattern of existential (there exists) and universal (for all) quantifiers

Complexity Quiz:

What is the complexity of the following problems?

- Minimize a deterministic FSM
 - $\text{DTIME}(k \cdot n \log n)$ $k = \# \text{symbols}$, $n = \# \text{states}$
- Minimize a non-deterministic FSM
 - PSPACE !
- Solving Generalized Chess
 - EXPTIME-complete
- Deciding if two regular expressions are equivalent
 - Four regex operators: (ab) , $(a|b)$, (a^*) , $(a\{2\})$
 - EXPSPACE-complete (NEXPTIME-complete without a^*)
- Solving the boolean satisfiability (SAT) problem
 - NP-complete
- Solving the boolean quantified formula problem
 - PSPACE-complete
- LZW compression algorithm
 - given strings s and t , will compressing s with an LZ78 method add t to the dictionary?
 - P-complete (so it is not in NC and therefore cannot be parallelized in polylogarithmic time!)
- Testing if a given integer is prime
 - Proven to be in P in 2002 by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena (AKS primality test)
 - Before that it was suspected not be in P, but was known to be in NP, Co-NP, BQP (Shor's Algorithm), and BPP.
- For a given finite set of numbers: Exists a subset so that the sum is in a given interval?
 - NP-complete
- Sorting by comparing elements
 - $\text{DTIME}(n \log n)$
- Building a binary heap
 - $\text{DTIME}(n)$
- Linear Programming
 - $\text{DTIME}(L \cdot n^{3.5})$ for n vars with L bits, P-complete
- Matrix-Matrix-Multiply
 - for $n \times n$ matrices: $\log_2(7) = 2.807..$
 - $\text{DTIME}(n^{2.807..})$ (Strassen, 1969)
 - $\text{DTIME}(n^{2.2374})$ (Strothers, 2011)
 - $\text{DTIME}(n^{2.23728642})$ (Williams, 2011)
 - $\text{DTIME}(n^{2.23728639})$ (Ge Gall, 2014)

15 Minutes Break

Optimizing P Algorithms

- Usually the worst case is optimized
- Often best case \approx avg. case \approx worst case
- Usually easy to create test data
- Examples:
 - Sorting, Linear Algebra
 - Regex matching, Compression

Optimizing NP+ Algorithms

- Worst case is usually out of reach
- Good algorithms find “loopholes” in the specific problem instance
- Hard to test from artificial data, need “industrial” data sets for performance evaluation.
- Hard to predict in advance how an algorithm will perform on a new data set.
- Examples:
 - SAT, QBF

Solvers for hard problems

- Many instances of hard problems are simple!
 - It is possible to write solvers for specific hard problems, manually implementing handling of all “known loopholes” in the specific problem set.
Example:
 - The subset-sum-problem (aka knapsack problem) is hard
 - Paying in cash money is easy with real-world currencies
- But what if we convert all NP-complete problems to the SAT problem?
 - Are the “loopholes” preserved?
 - Is it still possible to solve the easy instances of a hard problem efficiently after conversion to CNF?
 - In surprisingly many cases the answer is Yes!

SAT Solvers

- SAT solvers solve a SAT problem
 - Input in CNF (common format: DIMACS)
 - When sat: solver provides a model: The assignments that satisfies the formula
 - Some solvers can provide kernels for unsat
- Many solvers support incremental solving:
 - Pushing/popping of clauses (via different mechanisms)
 - Solving second problem using learned clauses from first version of the problem
- Can't compare solvers using worst-case performance
 - Annual competitions using industrial and synthetic DIMACS files

SMT Solvers

- SMT = SAT Modulo Theories
 - SAT Solver + Additional higher-level functions (“Theories”)
- Similar in functionality to SAT solvers, but higher-level problem descriptions
 - Structure of original problem is preserved much better
 - Some SMT solvers support quantifiers, making them QBF solvers
 - Some theories add infinite state, further extending the domain of SMT solvers
- Common input language for SMT solvers:
 - The SMT-LIB Language (currently in version 2.5)
 - Annual competitions with industrial and artificial problem sets
- Different “Theories” like Integers, Bit-Vectors, Arrays, Reals, etc..
 - A set of theories that might be used in conjunction is called a “Logic”
 - Logic used most often for BMC using Yosys:
 - QF_AUFBV (Quantifier-free, Arrays, Uninterpreted functions, Bit-Vectors)
 - Yosys benchmarks will be included in 2016 SMT competition

Formal Verification of digital Circuits: BMC and Temporal Induction

- Bounded Model Checking (BMC):

- Model of the circuit for N time steps
- Constraints for initial conditions
- Constraints for violated asserts
- When sat:
 - assert violated within N time steps
 - model contains counter-example

← Yosys: sat -seq N

- Temporal Induction

- Proving properties hold forever: Try induction!
- Model of the circuit for N+1 time steps
- Constraints for met asserts in first N time steps
- Constraints for violated asserts in last time step
- When unsat: Unbounded proof of properties

← Yosys: sat -tempinduct

- More advanced algorithms find additional invariants to prove with a smaller N
 - For example IC3 (implemented in command pdr in ABC)

Formal Verification with Yosys

- Built-in equivalence checking framework (see `help equiv_*`)
- Creating miter circuits for equivalence or property checking:
 - Either solve with built-in solver or
 - Export as BLIF and solve with e.g. ABC
- SAT solving (built-in MiniSAT-based eager SMT solver, see `help sat`)
- Creating SMT-LIB 2.5 models for circuits and properties that can be used with external SMT solvers
 - BMC and temporal induction with `yosys-smtbmc`

Equivalence Checking

- The `equiv_*` commands in Yosys are for equivalence checking.
- This equivalence checker uses hints like net names to partition the circuits into to-be-proved equivalent subcircuits.
- This is extremely helpful for troubleshooting Yosys passes and/or perform pre-vs-post synthesis verification.
- The prover is capable of considering multiple time-steps (`equiv_simple -seq N`) and even perform temporal induction (`equiv_induct`).

```
module gold(input A, B, output Y);  
  wire T = -A;  
  assign Y = T + B;  
endmodule
```

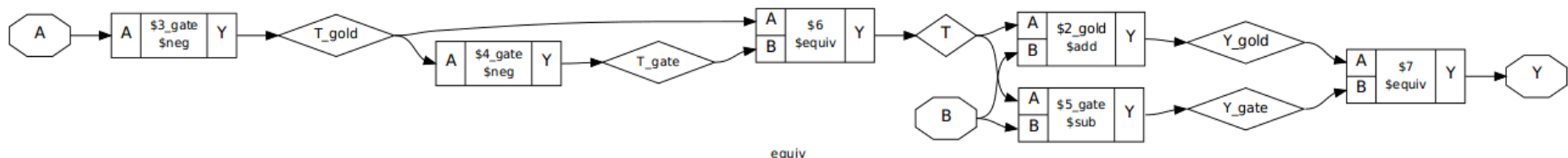
```
module gate(input A, B, output Y);  
  wire T = --A;  
  assign Y = T - B;  
endmodule
```

```
equiv_make gold gate equiv  
hierarchy -top equiv  
opt -purge; show  
equiv_simple  
equiv_status -assert
```

...

Found 2 \$equiv cells in equiv:

Of those cells 2 are proven and 0 are unproven.
Equivalence successfully proven!



Verilog asserts

- Yosys supports SystemVerilog assert and assume:

In module context:

```
assert property (<expression>);
```

```
assume property (<expression>);
```

In module context:

```
assert (<expression>);
```

```
assume (<expression>);
```

```
module example_assert(A);
    signed input  [31:0] A;
    signed wire   [31:0] B;
    assign B = A < 0 ? -A : A;
    assert property (B >= 0);
endmodule
```

```
read_verilog -sv example_assert.v
hierarchy -top example_assert
proc; opt -keepdc
sat -prove-asserts -show-inputs
```

```
$ yosys example_assert.y
```

...

Solving problem with 845 variables and 2305 clauses..

SAT proof finished - model found: FAIL!

[illegible]

Signal Name	Dec	Hex	Bin
\A	-2147483648	80000000	10000000000000000000000000000000

Miter circuits

- Some tools (e.g. ABC) operate on **miter circuits**.
 - Miters are circuits with a **single output**
 - that is **asserted when the property is violated**
- The miter command can be used to create miters
 - For **properties** using assert and/or assume statements
 - For **equivalence** of two circuits

```
read_verilog -sv example_miter.v
hierarchy; proc; opt; memory; opt
miter -assert main; techmap; opt
write_blif example_miter.blif
```

```
module main(input clk, output [63:0] state);
    reg [63:0] state = 123456789;

    function [63:0] xorshift64star;
        input [63:0] current_state;
        begin
            xorshift64star = current_state;
            xorshift64star = xorshift64star ^ (xorshift64star >> 12);
            xorshift64star = xorshift64star ^ (xorshift64star << 25);
            xorshift64star = xorshift64star ^ (xorshift64star >> 27);
            xorshift64star = xorshift64star * 64'd 2685821657736338717;
        end
    endfunction

    always @(posedge clk)
        state <= xorshift64star(state);

    assert property (state != 0);
endmodule
```

```
$ yosys-abc -c 'read_blif example_miter.blif; strash; pdr'
```

...

```
Invariant F[3] : 1 clauses with 64 flops (out of 64)
```

```
Verification of invariant with 1 clauses was successful. Time = 0.01 sec
```

```
Property proved. Time = 0.28 sec
```

Yosys sat command

- The Yosys sat command provides access to the **built-in SAT solver** framework
 - Based on **MiniSAT** SimpSolver
 - Essentially an **eager SMT solver**
- Bounded Model Checking (BMC):
 - `sat -seq 50 -prove-asserts -set-assumes`
- Temporal Induction Proofs:
 - `sat -tempinduct -prove never_one 0`
- Writing traces as VCD files:
 - `sat ... -dump_vcd <vcd_filename> ...`
- Writing SAT problem in DIMACS format:
 - `sat ... -dump_cnf <dimacs_filename> ...`
- Interactive Troubleshooting:
 - `sat -seq 15 -set foo 23 -set-at 10 never_one 1 -show bar`

Temporal Induction Demo

- `yosys tempind01.y`
 - How does the induction length change when the alternate assert in `tempind01.y` is enabled?
 - Why is it so?
- `yosys tempind02.y`
 - How does the induction length change when some of the asserts are disabled?
 - Why is it so?

SMT-LIB: General Syntax

- Lisp S-expression -like syntax
- First (non-option) statement must declare logic:
`(set-logic QF_UF)`
- Since SMT-LIB 2.5 the solver may also auto-detect the logic:
`(set-logic ALL)`
- A “variable” is declared as uninterpreted function without arguments:
`(declare-fun myvar () Bool)`
- Constraints can be declared as asserts:
`(assert (or myvar (not myvar)))`
- Use `(check-sat)` to run the solver

(The information on this slide is sufficient to encode any CNF as SMT-LIB file.)

SMT-LIB: Options

- Disable printing of (success) after each statement:
 `(set-option :print-success false)`
 (this is the default for many solvers)
- Enable generation of modules:
 `(set-option :produce-models true)`
- Setting of random seed (optional):
 `(set-option :random-seed 42)`

SMT-LIB: Inspecting the Model

- Using `(get-value)`
- Requires option `:produce-models`
- Only when `(check-sat)` returned `sat`
- Works with every quantifier-free closed expression

```
$ cat smtex01.smt2
(set-option :produce-models true)
(set-logic QF_UF)
(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(assert (or a b))
(assert (and b (not c)))
(assert (or c (not a)))
(check-sat)
(get-value (a b c))
(get-value ((xor a b)))
```

```
$ z3 -smt2 smtex01.smt2
sat
((a false)
 (b true)
 (c false))
(((xor a b) true))
```

SMT-LIB: Incremental Solving

- Create new context for assertions:
(push 1)
- Destroy innermost context for assertions:
(pop 1)
- Use a larger integer to push/pop multiple contexts at once.

```
$ cat smtex02.smt2
(set-option :produce-models true)
(set-logic QF_UF)
(declare-fun a () Bool)
(declare-fun b () Bool)
(assert (xor a b))
(check-sat)
```

```
(push 1)
(assert (and a b))
(check-sat)
```

```
(pop 1)
(assert (or a b))
(check-sat)
```

```
$ z3 -smt2 smtex02.smt2
sat
unsat
sat
```

SMT-LIB: Functions, Constants

- Bool constants: `true` and `false`
 - Some theories define additional constants

- Defining a function:

```
(define-fun
  my_and                ; function name
  ((a Bool) (b Bool)) ; arguments
  Bool                  ; return type
  (and a b))            ; expression
```


SMT-LIB: Custom Sorts

- Types are called sorts in SMT-LIB nomenclature
- Use `(declare-sort my_sort_name 0)` to declare custom sorts, for example:

```
$ cat smtex03.smt2
(set-option :produce-models true)

(set-logic QF_UF)
(declare-sort color 0)
(declare-fun r () color) ; red
(declare-fun g () color) ; green
(declare-fun b () color) ; blue
(assert (distinct r g b))

(declare-fun at () color) ; austria
(declare-fun de () color) ; germany
(declare-fun li () color) ; liechtenstein
(declare-fun it () color) ; italy
(declare-fun ch () color) ; switzerland

(assert (or (= at r) (= at g) (= at b)))
(assert (or (= de r) (= de g) (= de b)))
(assert (or (= li r) (= li g) (= li b)))
(assert (or (= it r) (= it g) (= it b)))
(assert (or (= ch r) (= ch g) (= ch b)))

(assert (distinct at de))
(assert (distinct at li))
(assert (distinct at it))
(assert (distinct at ch))
(assert (distinct ch de))
(assert (distinct ch li))
(assert (distinct ch it))

(check-sat)
(get-value (r g b at de li it ch))

$ z3 -smt2 smtex03.smt2
sat
((r color!val!0)
 (g color!val!1)
 (b color!val!2)
 (at color!val!1)
 (de color!val!2)
 (li color!val!2)
 (it color!val!2)
 (ch color!val!0))
```

SMT-LIB: Theory of Bit-Vectors

- Declaring a Bit-Vector:
`(declare-fun my_bv_var () (_ BitVec 16))`
- Concatenation and extraction:
`(concat a b)`
`((_ extract 11 8) c)`
- Operations on Bit-Vectors, for example:
`(bvadd a b)`
- Bit-Vector constants:
`#b1101001`
- Conversion from/to Bool:
`(ite bool_expr #b1 #b0)`
`(= bv_expr #b1)`

SMT-LIB: Theory of Arrays

- Declaring an Array (16 elements, 8 bits each):

```
(declare-fun my_array () (Array (_ BitVec 4) (_ BitVec 8)))
```

- Accessing an Array element:

```
(select my_array #b0011)
```

- Creating a modified version of an Array:

```
(store my_array #b1010 #b00110011)
```

- Can be used to model memory efficiently!

10 Minutes Break

SMT Solvers for QF_AUFBV

- The logic QF_AUFBV is required for our verification flow. The following solvers support this logic (listed from fastest to slowest, results may vary).
- Yices2
 - “This EULA permits use of the software **only for projects that do not receive external funding** other than government research grants and contracts. Any **other use requires a commercial license.**”
- Z3
 - “Z3 is licensed under the **MIT license.**” This is the default solver for yosys - smt bmc.
- CVC4
 - “The source code of CVC4 is open and available to students, researchers, software companies, and everyone else to study, to modify, and to redistribute original or modified versions; distribution is under the terms of the **modified BSD** license. However, CVC4 can be configured (and is, by default) to link against some **GPLed libraries**, [...]”
- MathSAT5
 - “MathSAT5 is available for **research and evaluation purposes only**. It can not be used in a commercial environment, particularly as part of a commercial product, without written permission. [...]”

Yosys SMT2 Back-End (1/2)

- The Yosys SMT2 back-end converts digital designs into SMT2 code
 - Use option `-bv` to enable support for theory of bit-vectors (for RTL cells)
 - Use option `-mem` to enable support for theory of arrays (for memories)
- The SMT2 back-end defines one sort for each module that represents the state of the module at one point in time
 - Example: `module main` \rightarrow `main_s`
 - Usage: `(declare-fun s01 () main_s)`
- It also defines a function for each net
 - By default only for some wires, use for example `-wires` or `-regs` for better coverage
 - Example: `main.foobar` in `s01` \rightarrow `(|main_n foobar| s01)`
- And it defines a function for state transitions
 - Example: `(assert (main_t s01 s02))`

Yosys SMT2 Back-End (2/2)

- Additional functions for module states:
 - Initial state: `(main_i s01)`
 - Assertions met: `(main_a s01)`
 - Assumptions met: `(main_u 01)`
- Access to memories:
 - Get array for memory `mymem`: `(|main_m mymem| s01)`
- More information: `help write_smt2`

Yosys-SMTBMC Tool

- Extending the SMT2 code created by the SMT2 back-end to useful proofs is not hard, but sometimes inconvenient.
- The tool `yosys-smtbmc` can read the SMT2 files created by `write_smt2` and to perform
 - Bounded Model Checking and
 - Temporal Induction Proofs
- It can use various different SMT solvers (default: z3)
- Can dump traces as SMT2 files for performance comparisons
- Can write counter-examples as VCD files

Yosys-SMTBMC Examples

- `bash tempind02.sh`
 - `yosys-smtbmc` as replacement for `sat` command
- `make check` in PicoRV32
 - Finally a non-trivial example:
Property checking in PicoRV32 RISC-V CPU

Custom proofs with smtio.py

- `smtio.py` is a python library for remote controlling SMT solvers and building proofs around the SMT2 files written by Yosys
 - `yosys-smtbmc` is just a small script using `smtio.py`
- PicoRV32 CPU
 - `sync.sh` / `async.sh` bounded equivalence checker for different configurations of the CPU core (`async.sh` for cores with different cycles per instructions)
- PonyLink Slave Reset Verification
 - PonyLink is a chip-to-chip master-slave communications core using a single half-duplex communications channel
 - A custom `smtio.py` proof is used to prove that
 - The slave will always stop sending after a short period of time, regardless of its initial state
 - Resetting the slave over the link will always succeed, regardless of the slaves initial state

Yosys SMT2 Todos

(as of 2015-10-15)

- Function: module state → large Bit-Vector
 - As witness for distinct states
- Support for hierarchical designs
 - With a per-state assert on the top-level module
 - Most of the work is in `smtio.py` for VCD writing
- Documentation and Examples
 - This presentation is a start
 - Finding non-trivial small examples is hard!
- Related: Better SystemVerilog / PSL support

Questions?

P, NP, PSPACE,
Deterministic, Nondeterministic,
FSMs, Turing Machines, Decidability,
NP-hard, NP-complete,
SAT, QBF

SAT + SMT Solvers,
CNF, Theories, BMC,
Temporal Induction

Formal Verification,
Equivalence Checking,
Verilog assert / assume,
Miter circuits

QF_AUFBV Solvers,
Yosys SMT2 Back-end,
Yosys-SMTBMC, smtio.py

SMT2-LIB, Theories, Logics,
Sorts, Uninterpreted Functions,
Bit-Vectors, Arrays