Bit Vector Decision Procedures

A Basis for Reasoning about Hardware & Software

Randal E. Bryant
Carnegie Mellon University

`http://www.cs.cmu.edu/~bryant`

Collaborators

Sanjit Seshia, Bryan Brady
- UC Berkeley

Daniel Kroening
- ETH Zurich

Joel Ouaknine
- Oxford University

Ofer Strichman
- Technion

Randy Bryant
- Carnegie Mellon

3 continents
4 countries
5 institutions
6 authors

Motivating Example #1

```
int abs(int x) {
  int mask = x>>31;
  return (x ^ mask) + ~mask + 1;
}
```

```
int test_abs(int x) {
  return (x < 0) ? -x : x;
}
```

Do these functions produce identical results?

Strategy

- Represent and reason about bit-level program behavior
- Specific to machine word size, integer representations, and operations

Motivating Example #2

```
void fun() {
  char fmt[16];
  fgets(fmt, 16, stdin);
  fmt[15] = '\0';
  printf(fmt);
}
```

Is there an input string that causes value 234 to be written to address $a_4a_3a_2a_1$?

Answer

- **Yes: "$a_1a_2a_3a_4$%230g%n"**
  - Depends on details of compilation
- **But no exploit for buffer size less than 8**
- **[Ganapathy, Seshia, Jha, Reps, Bryant, ICSE ' 05]**

Motivating Example #3

```
bit[W] popSpec(bit[W] x)
{
  int cnt = 0;
  for (int i=0; i<W; i++) {
    if (x[i]) cnt++;
  }
  return cnt;
}
```

```
bit[W] popSketch(bit[W] x)
{
  loop (??) {
    x = (x&??) + ((x>>??)&??);
  }
  return x;
}
```

Is there a way to expand the program sketch to make it match the spec?

Answer

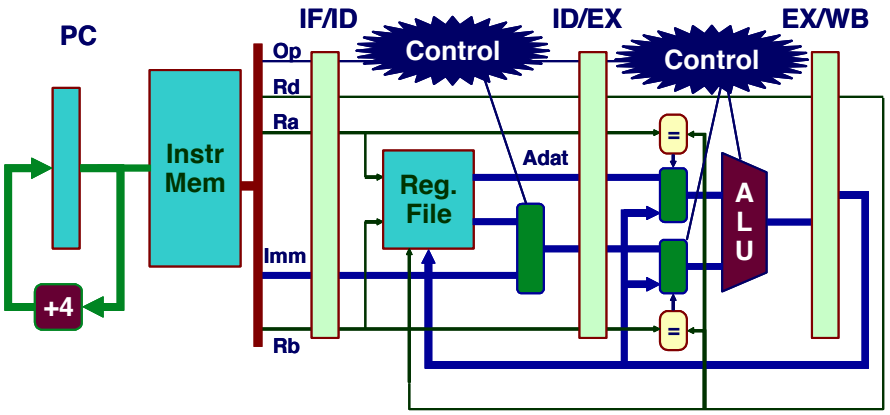
- **W=16:**

```
x = (x&0x5555) + ((x>>1)&0x5555);
x = (x&0x3333) + ((x>>2)&0x3333);
x = (x&0x0077) + ((x>>8)&0x0077);
x = (x&0x000f) + ((x>>4)&0x000f);
```

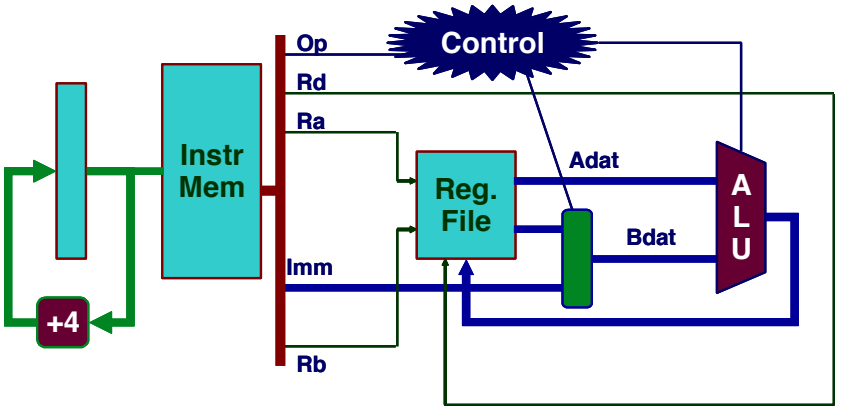
- **[Solar-Lezama, et al., ASPLOS '06]**

Motivating Example #4

Pipelined Microprocessor



Sequential Reference Model



Is pipelined microprocessor identical to sequential reference model?

Strategy

- **Automatically generate abstraction function from pipeline to program state [Burch & Dill, CAV ' 94]**
- **Represent machine instructions, data, and state as bit vectors**
  - Compatible with hardware description language representation

Task

Bit Vector Formulas

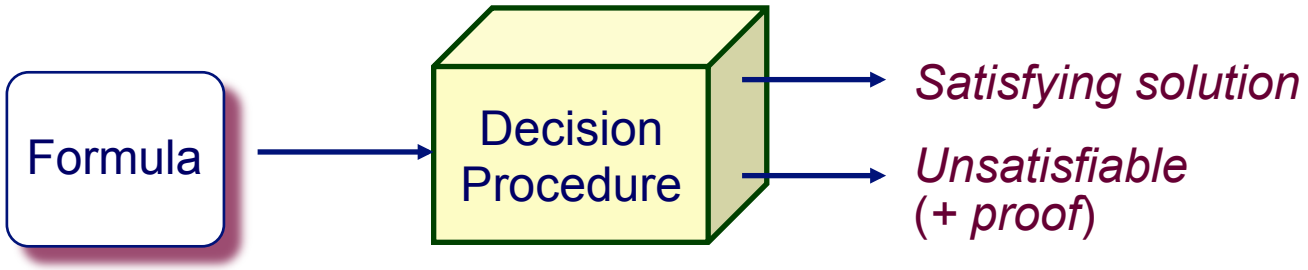
- **Fixed width data words**
- **Arithmetic operations**
  - E.g., add/subtract/multiply/divide & comparisons
  - Two's complement, unsigned, …
- **Bit-wise logical operations**
  - E.g., and/or/xor, shift/extract and equality
- **Boolean connectives**

Reason About Hardware & Software at Bit Level

- **Formal verification**
- **Security analysis**
- **Test & program generation**
  - What function arguments will cause program to take specified branch?

Decision Procedures

- **Core technology for formal reasoning**



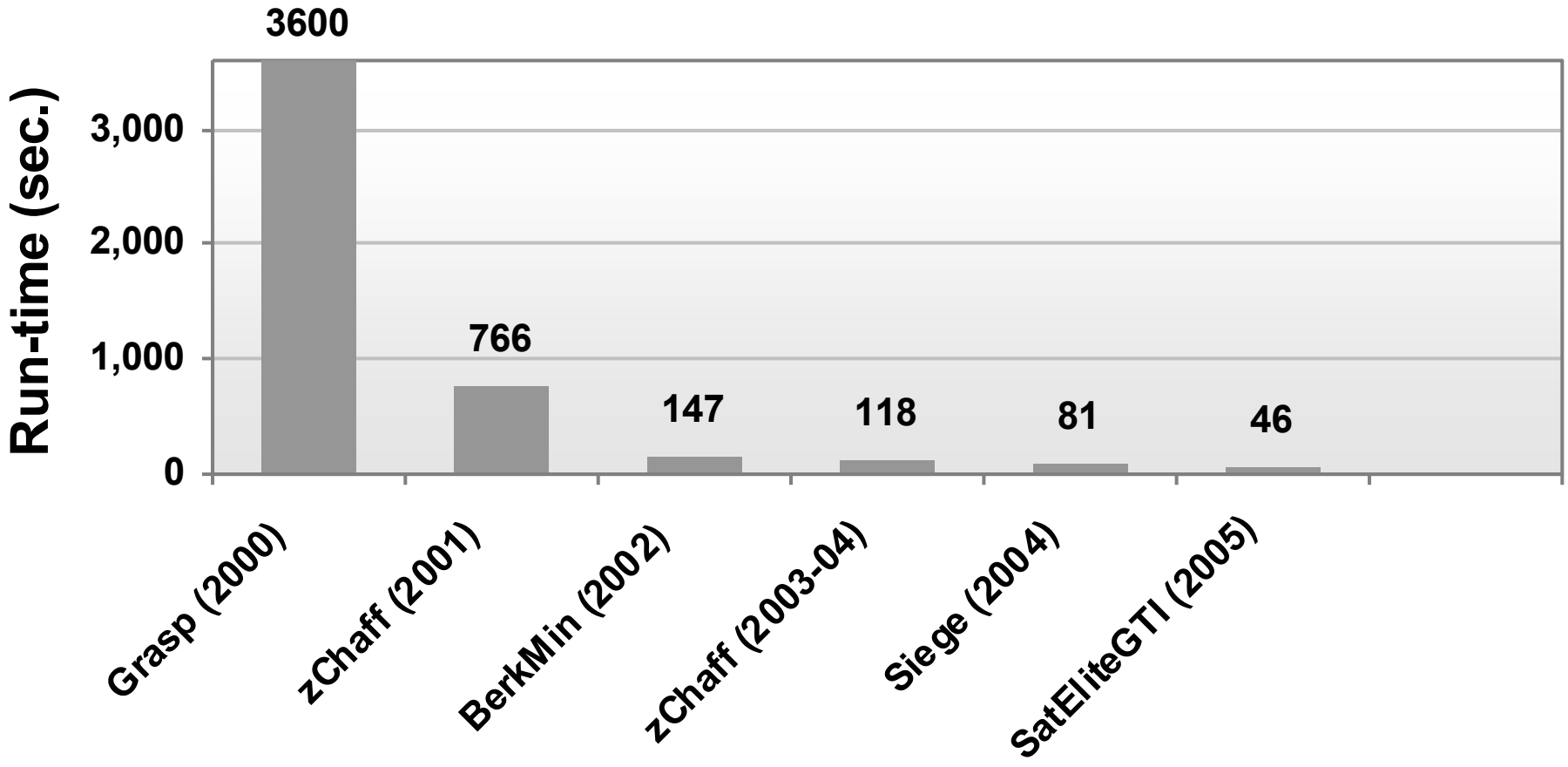
Boolean SAT

- **Pure Boolean formula**

SAT Modulo Theories (SMT)

- **Support additional logic fragments**
- **Example theories**
  - Linear arithmetic over reals or integers
  - Functions with equality
  - *Bit vectors*
  - Combinations of theories

Recent Progress in SAT Solving



BV Decision Procedures: Some History

B.C. (Before Chaff)

- String operations (concatenate, field extraction)
- Linear arithmetic with bounds checking
- Modular arithmetic

SAT-Based "Bit Blasting"

- Generate Boolean circuit based on bit-level behavior of operations
- Convert to Conjunctive Normal Form (CNF) and check with best available SAT checker
- Handles arbitrary operations
- Effective in many applications
  - CBMC [Clarke, Kroening, Lerda, TACAS '04]
  - Microsoft Cogent + SLAM [Cook, Kroening, Sharygina, CAV '05]
  - CVC-Lite [Dill, Barrett, Ganesh], Yices [deMoura, et al], STP

Research Challenge

Is there a better way than bit blasting?

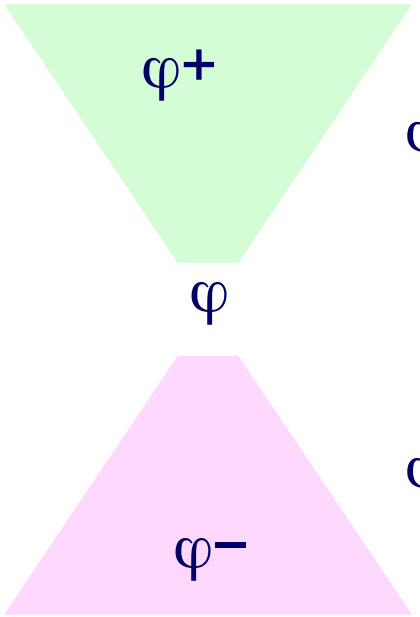
Requirements

- Provide same functionality as with bit blasting
- Find abstractions based on word-level structure
- Improve on performance of bit blasting

A New Approach

- [Bryant, Kroening, Ouaknine, Seshia, Stichman, Brady, TACAS ' 07]
- Use bit blasting as core technique
- Apply to simplified versions of formula
- Successive approximations until solve or show unsatisfiable

Approximating Formula



Overapproximation

$\varphi \Rightarrow \varphi^+$

More solutions:
If unsatisfiable, then so is φ

Original Formula

Underapproximation

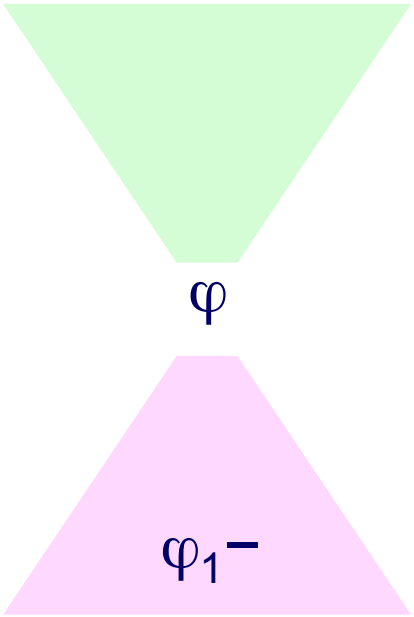
$\varphi^- \Rightarrow \varphi$

Fewer solutions:
Satisfying solution also satisfies φ

Example Approximation Techniques

- **Underapproximating**
  - Restrict word-level variables to smaller ranges of values
- **Overapproximating**
  - Replace subformula with Boolean variable

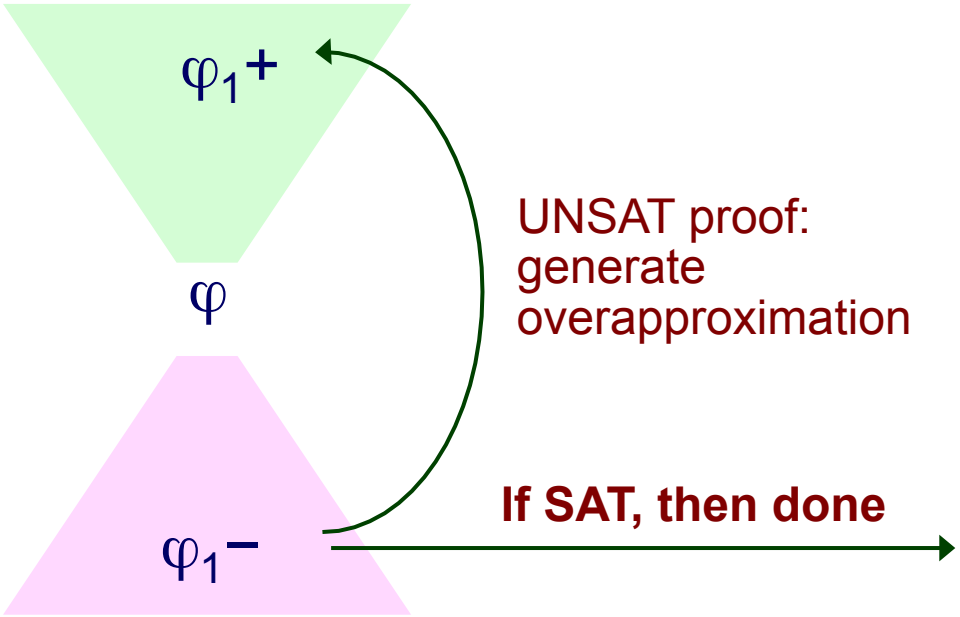
Starting Iterations



Initial Underapproximation

- (Greatly) restrict ranges of word-level variables
- Intuition: Satisfiable formula often has small-domain solution

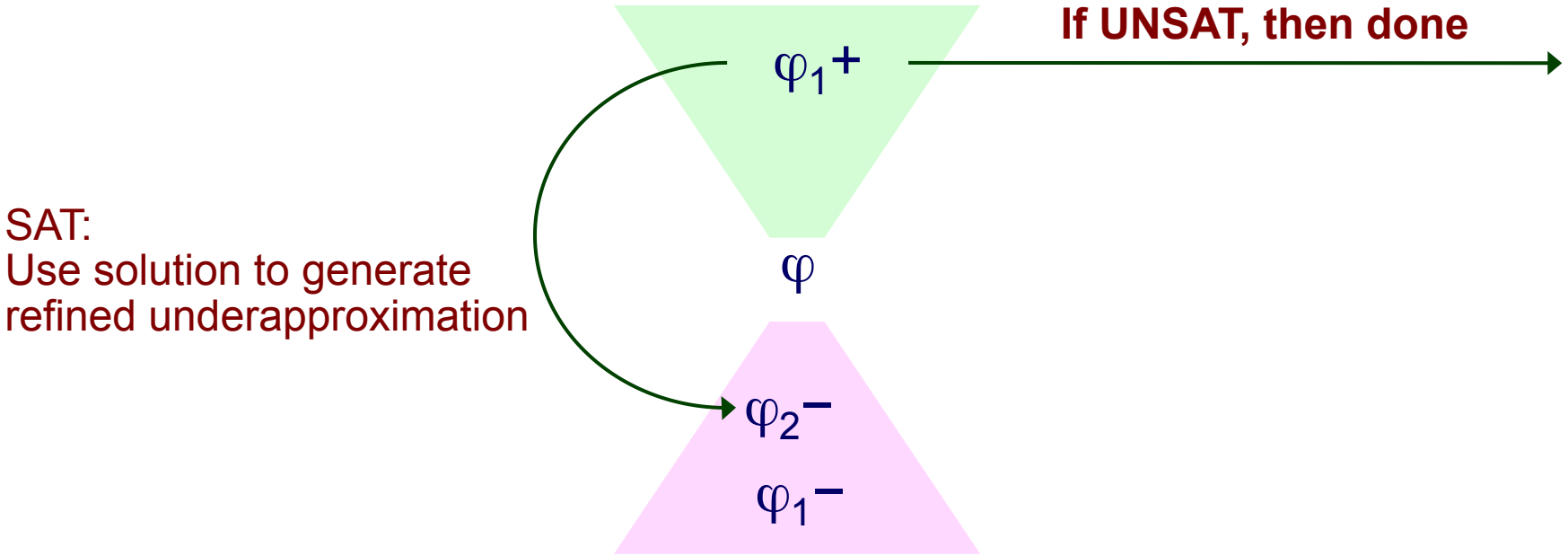
First Half of Iteration



SAT Result for φ_1-

- **Satisfiable**
  - Then have found solution for φ
- **Unsatisfiable**
  - Use UNSAT proof to generate overapproximation φ_1+
  - (Described later)

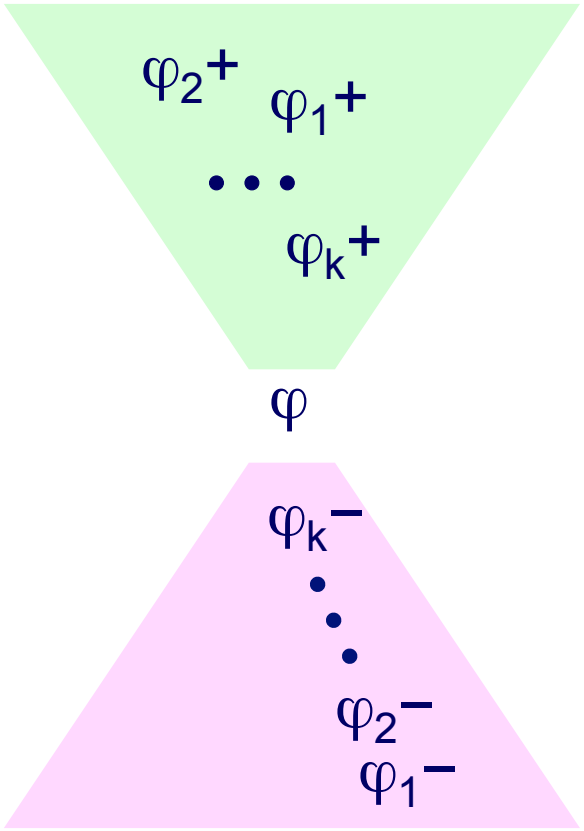
Second Half of Iteration



SAT Result for φ_1+

- **Unsatisfiable**
  - Then have shown φ unsatisfiable
- **Satisfiable**
  - Solution indicates variable ranges that must be expanded
  - Generate refined underapproximation

Iterative Behavior



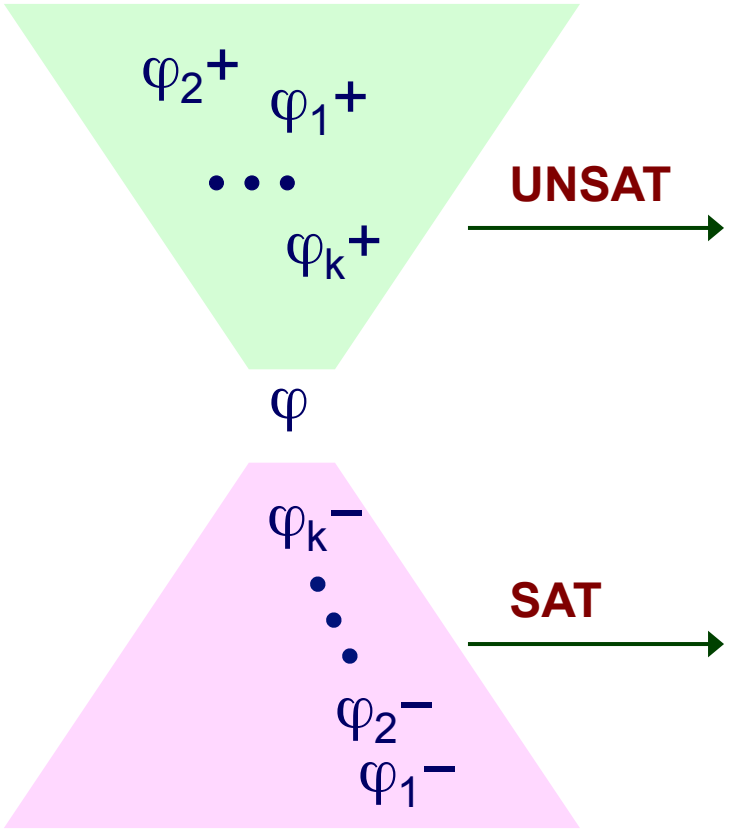
Underapproximations

- Successively more precise abstractions of φ
- Allow wider variable ranges

Overapproximations

- No predictable relation
- UNSAT proof not unique

Overall Effect



Soundness

- Only terminate with solution on underapproximation
- Only terminate as UNSAT on overapproximation

Completeness

- Successive underapproximations approach φ
- Finite variable ranges guarantee termination
  - In worst case, get $\varphi_k^- = \varphi$

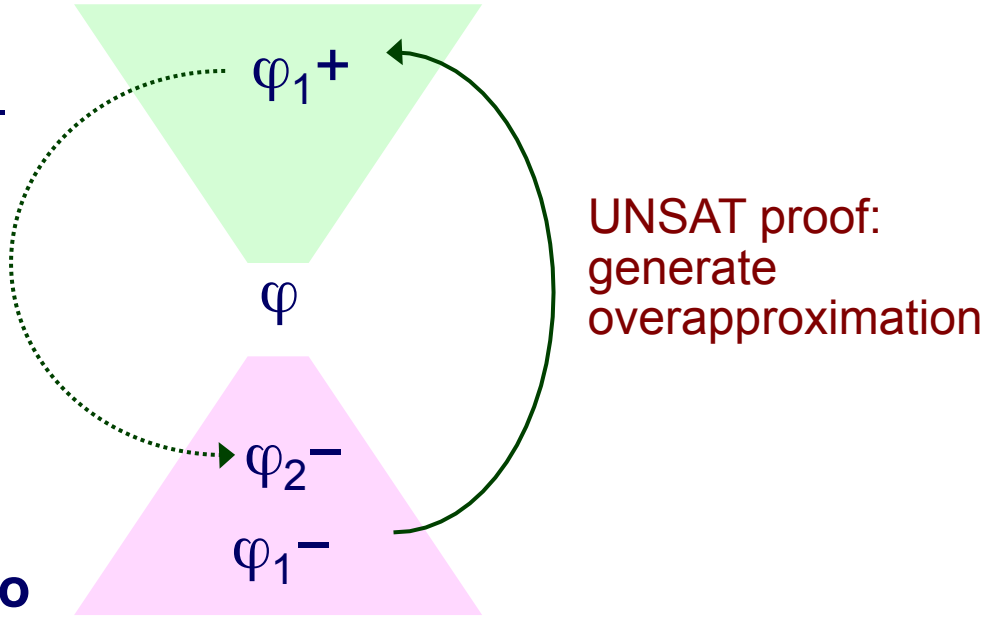
Generating Overapproximation

Given

- Underapproximation φ_1-
- Bit-blasted translation of φ_1- into Boolean formula
- Proof that Boolean formula unsatisfiable

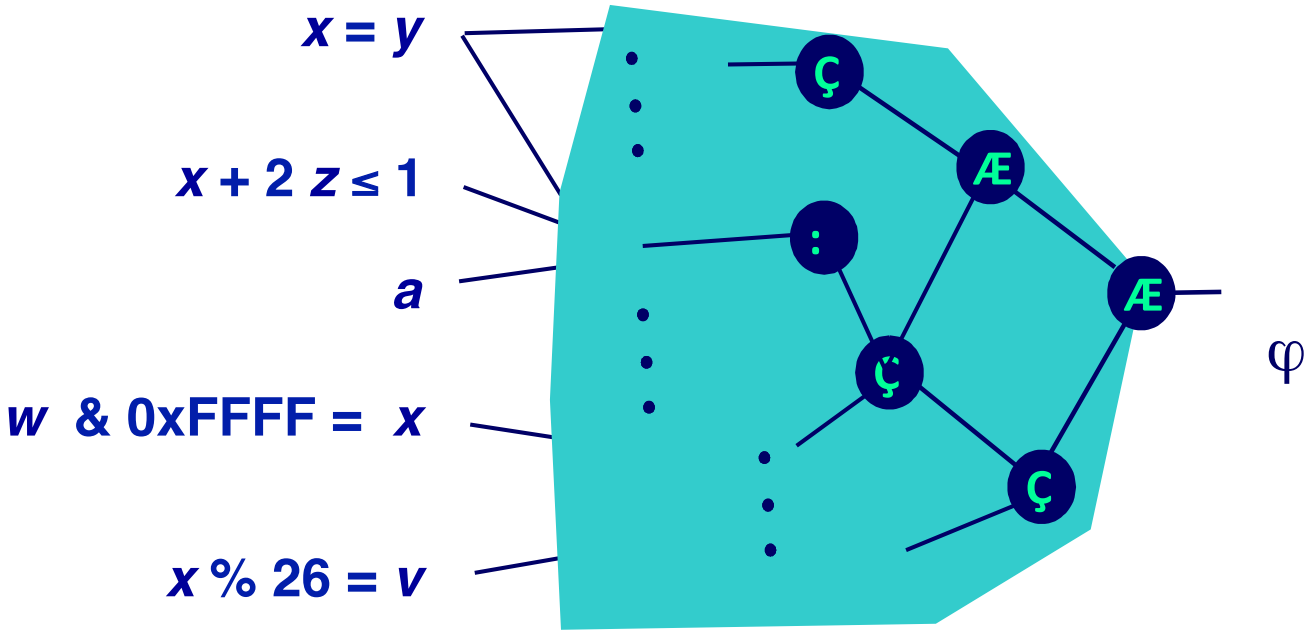
Generate

- Overapproximation φ_1+
- If φ_1+ satisfiable, must lead to refined underapproximation
  - Generate φ_2- such that $\varphi_1- \Rightarrow \varphi_2- \Rightarrow \varphi$

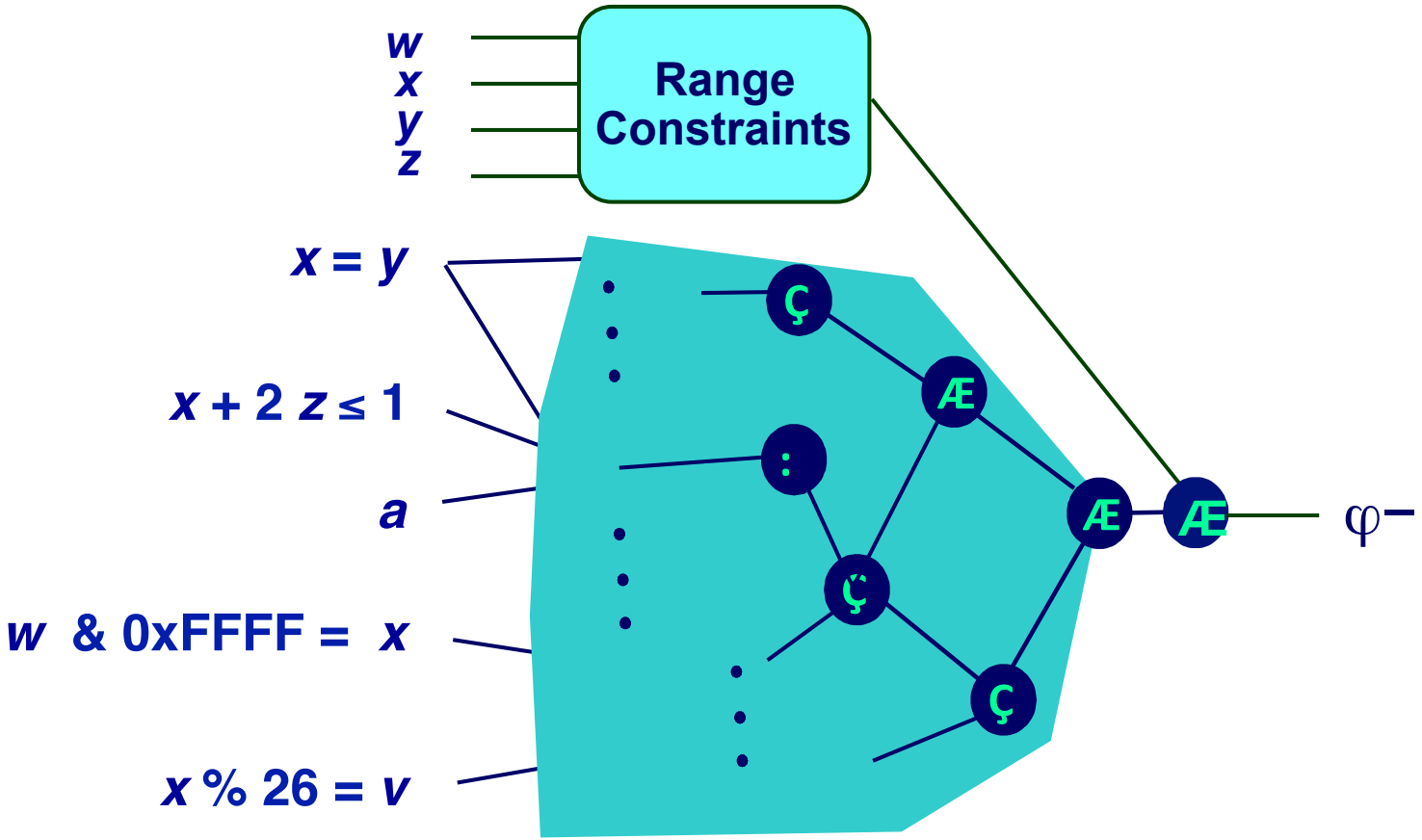


Bit-Vector Formula Structure

- DAG representation to allow shared subformulas



Structure of Underapproximation



- **Linear complexity translation to CNF**
  - Each word-level variable encoded as set of Boolean variables
  - Additional Boolean variables represent subformula values

Encoding Range Constraints

Explicit

- View as additional predicates in formula

$0 \le w < 8 \;\wedge\; -4 \le x < 4$

Implicit

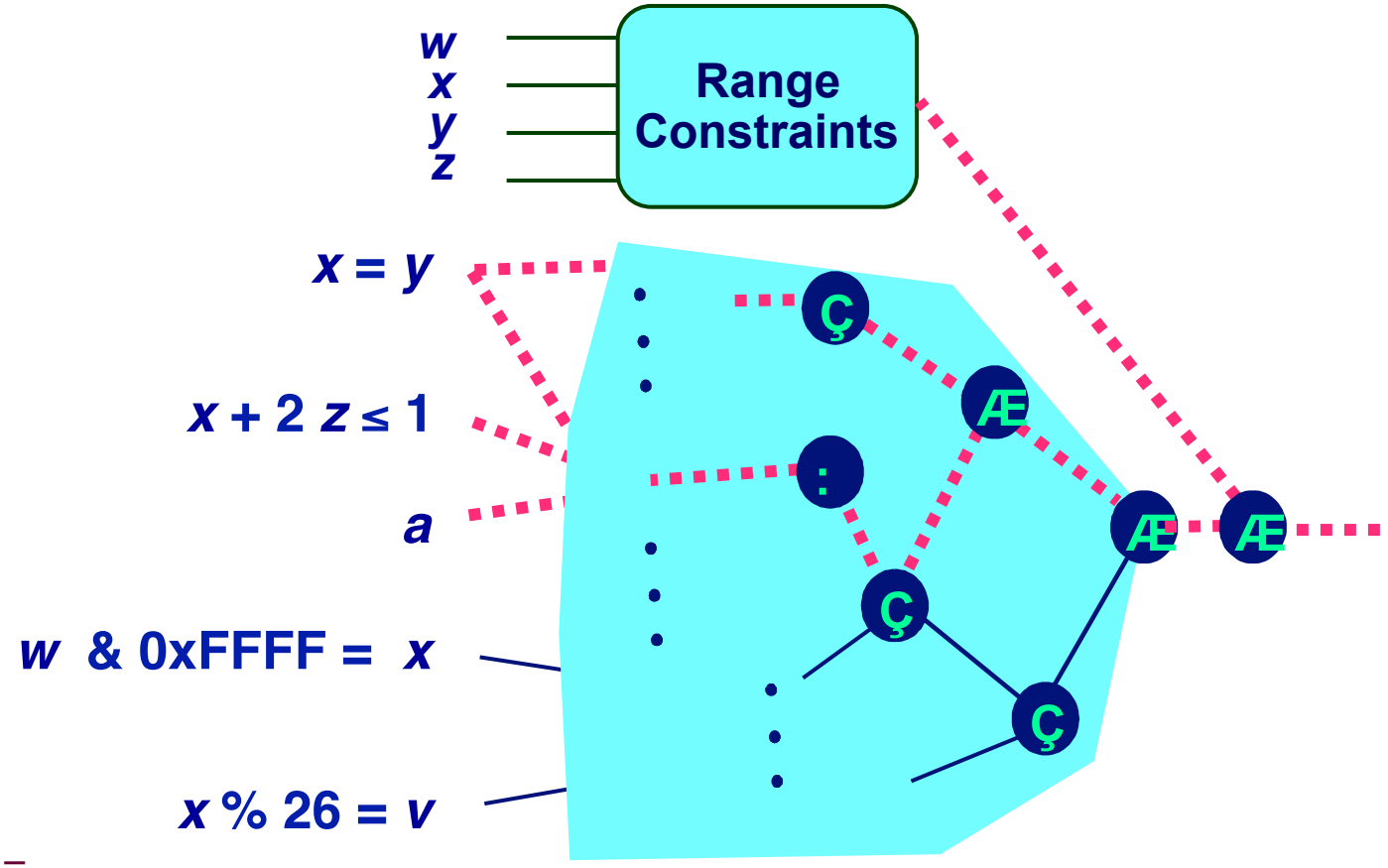
- Reduce number of variables in encoding

Constraint	Encoding
$0 \le w < 8$	$0\ 0\ 0 \cdots 0\ w_2 w_1 w_0$
$-4 \le x < 4$	$x_s x_s x_s \cdots x_s x_s x_1 x_0$

- Yields smaller SAT encodings

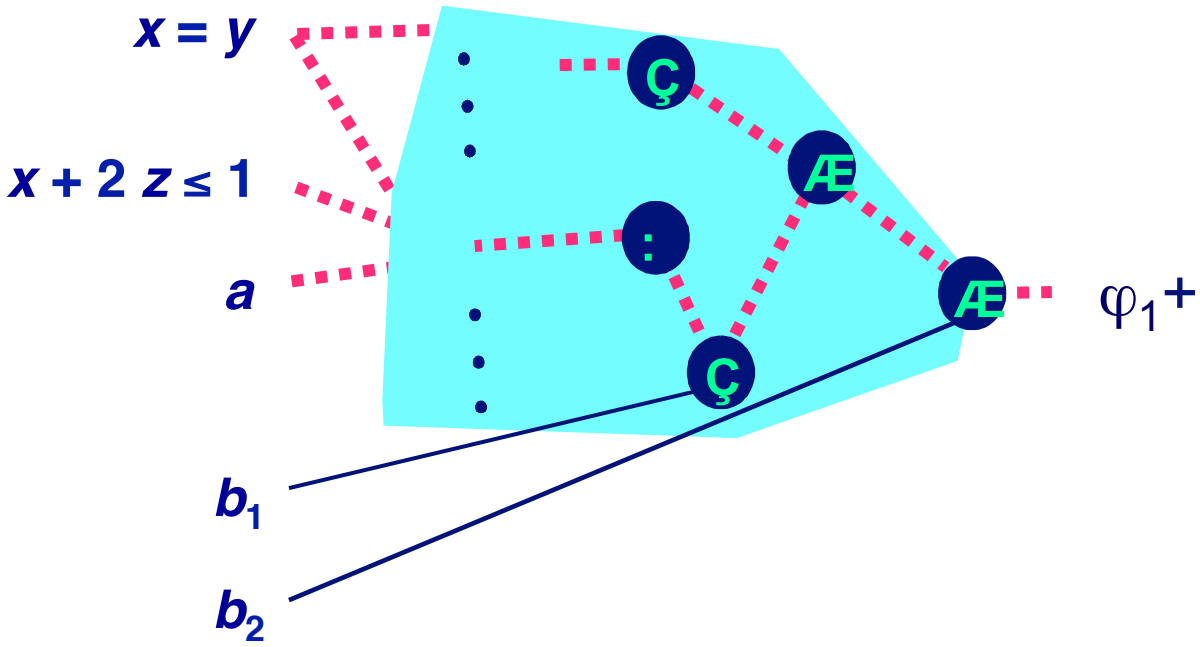
UNSAT Proof

- Subset of clauses that is unsatisfiable
- Clause variables define portion of DAG
- Subgraph that cannot be satisfied with given range constraints



Generated Overapproximation

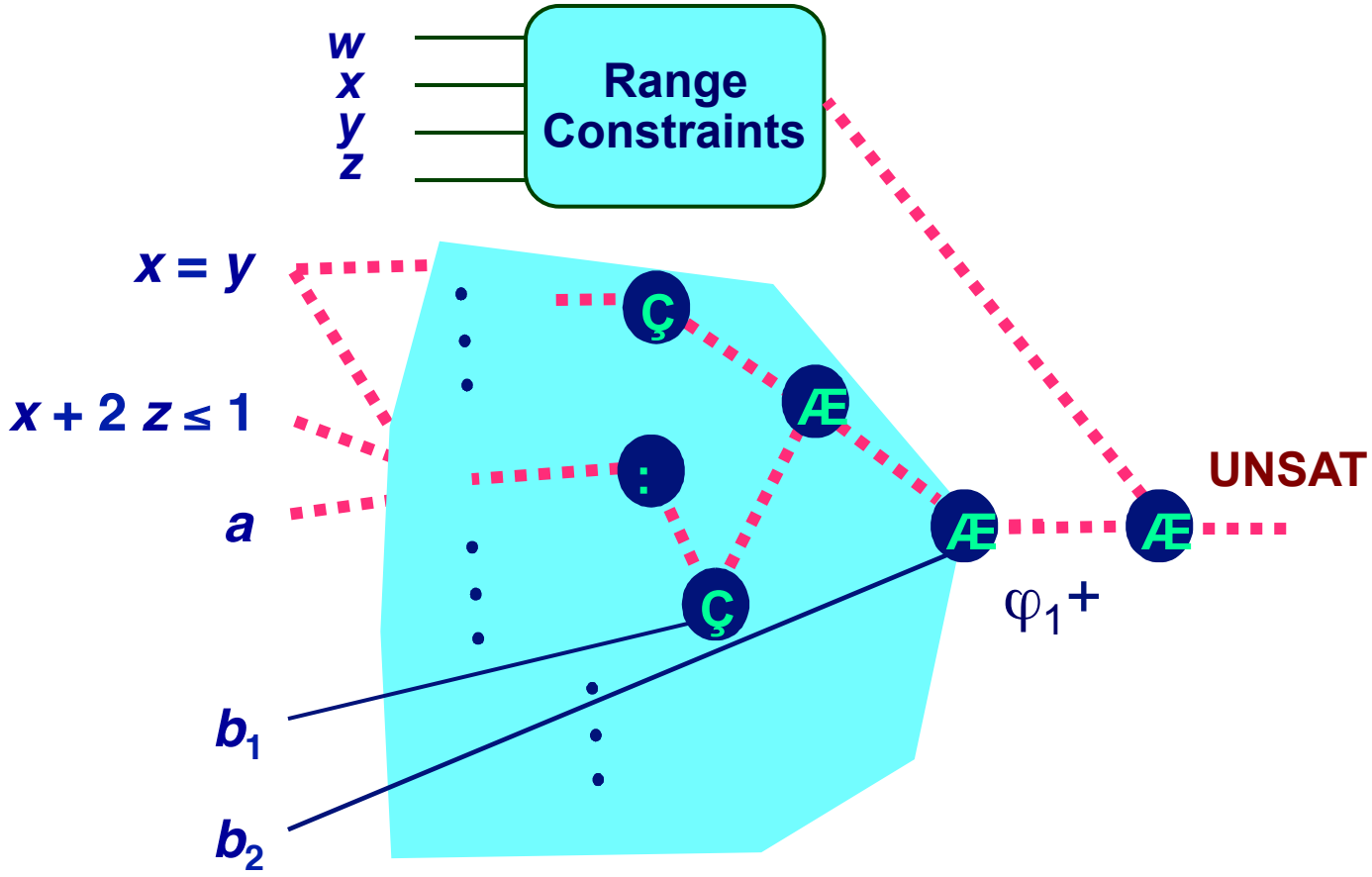
- Identify subformulas containing no variables from UNSAT proof
- Replace by fresh Boolean variables
- Remove range constraints on word-level variables
- Creates overapproximation
  - Ignores correlations between values of subformulas



Refinement Property

Claim

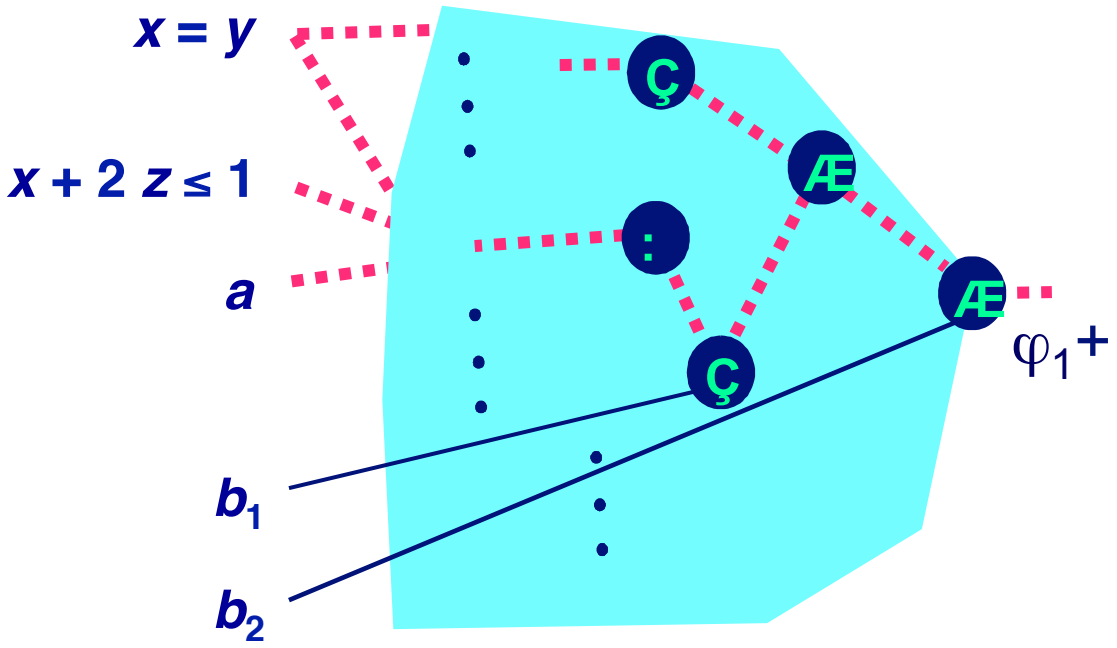
- φ_1+ **has no solutions that satisfy** φ_1-**'s range constraints**
  - Because φ_1+ contains portion of φ_1- that was shown to be unsatisfiable under range constraints



Refinement Property (Cont.)

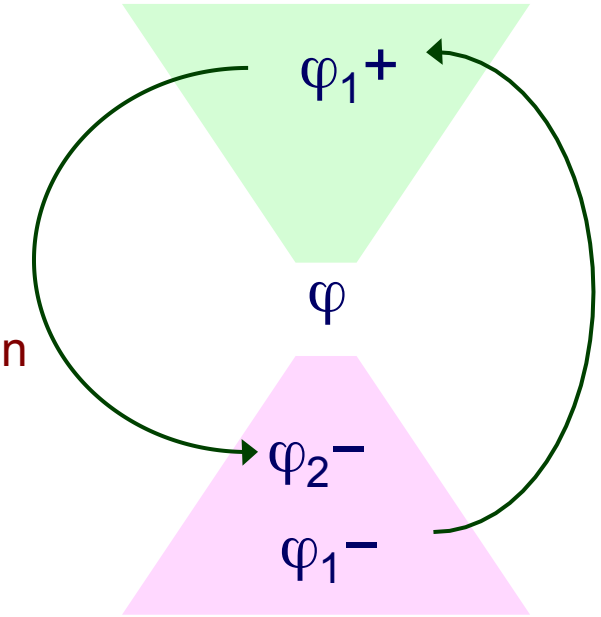
Consequence

- Solving φ_1+ will expand range of some variables
- Leading to more exact underapproximation φ_2-



Effect of Iteration

SAT:
Use solution to generate
refined underapproximation



UNSAT proof:
generate
overapproximation

Each Complete Iteration

- **Expands ranges of some word-level variables**
- **Creates refined underapproximation**

Approximation Methods

So Far

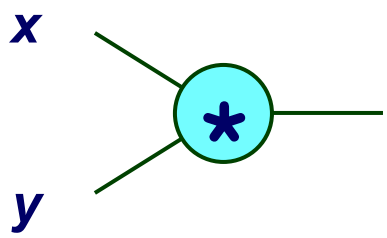
- **Range constraints**
  - Underapproximate by constraining values of word-level variables
- **Subformula elimination**
  - Overapproximate by assuming subformula value arbitrary

General Requirements

- **Systematic under- and over-approximations**
- **Way to connect from one to another**

Goal: Devise Additional Approximation Strategies

Function Approximation Example



		x		
		0	1	else
y	0	0	0	0
	1	0	1	x
	else	0	y	§

Motivation

- Multiplication (and division) are difficult cases for SAT

§: Prohibited

- Gives underapproximation
- Restricts values of (possibly intermediate) terms

§: $f(x,y)$

- Overapproximate as uninterpreted function f
- Value constrained only by functional consistency

Results: UCLID BV vs. Bit-blasting

Formula	Ans.	Bit-Blasting Run-time (sec.)			UCLID Run-time (sec.)			STP (sec.)	Yices (sec.)
		Enc.	SAT	Total	Enc.	SAT	Total		
Y86-std	UNSAT	17.91	TO	TO	23.51	987.91	**1011.42**	2083.73	TO
Y86-btnft	UNSAT	17.79	TO	TO	26.15	1164.07	**1190.22**	err	TO
s-40-50	SAT	6.00	33.46	39.46	106.32	10.45	116.77	**12.96**	65.51
BBB-32	SAT	37.09	29.98	67.07	19.91	1.74	**21.65**	38.45	183.30
rfunit_flat-64	SAT	121.99	32.16	154.15	19.52	1.68	**21.20**	873.67	1312.00
C1-P1	SAT	2.68	45.19	47.87	2.61	0.58	**3.19**	err	err
C1-P2	UNSAT	0.44	TO	TO	2.24	2.12	**4.36**	TO	TO
C3-OP80	SAT	14.96	TO	TO	14.54	349.41	**363.95**	TO	3242.43
egt-5212	UNSAT	0.064	0.003	0.067	0.163	0.001	0.164	0.018	**0.009**

[results on 2.8 GHz Xeon, 2 GB RAM]

- **UCLID always better than bit blasting**
- **Generally better than other available procedures**
- **SAT time is the dominating factor**

UCLID BV run-time analysis

Formula	Ans.	$\max_i\ s_i$	$\max_i\ w_i$	Num. Iter	$\max \frac{\lvert\overline{\phi}\rvert}{\lvert\phi\rvert}$	Speedup
Y86-std	UNSAT	4	32	1	0.18	2.06
Y86-btnft	UNSAT	4	32	1	0.20	$>$ 3.01
s-40-50	SAT	32	32	8	0.12	0.11
BBB-32	SAT	4	32	1	–	1.78
rfunit_flat-64	SAT	4	64	1	–	7.27
C1-P1	SAT	2	65	1	–	15.00
C1-P2	UNSAT	2	14	1	1.00	$>$ 825.69
C3-OP80	SAT	2	9	1	–	8.91
egt-5212	UNSAT	8	8	1	0.13	0.06

- **w_i: Maximum word-level variable size**
- **s_i: Maximum word-level variable instantiation**

- **Generated abstractions are small**
- **Few iterations of refinement loop needed**

Why This Work is Worthwhile

Realistic Semantic Model for Hardware and Software

- **Captures all details of actual operation**
  - Detects errors related to overflow and other artifacts of finite representation
- **Allows mixing of integer and bit-level operations**
- **Can capture many abstractions that are currently applied manually**

SAT-Based Methods Are Only Logical Choice

- **Bit blasting is only way to capture full set of operations**
- **SAT solvers are good & getting better**

Abstraction / Refinement Allows Better Scaling

- **Take advantage of cases where formula easily satisfied or disproven**

Future Work

Lots of Refinement & Tuning

- **Selecting under- and over-approximations**
- **Iterating within under- or over-approximation**
  - E.g., attempt to control variable ranges when overapproximating
- **Reusing portions of bit-blasted formulas**
  - Take advantage of incremental SAT

Additional Abstractions

- **More general use of functional abstraction**
  - Subsume use of uninterpreted functions in current verification methods